A CMOS DVD 4x Viterbi Detector: System Design and VLSI Implementation

by

Kenneth Chalmers

A thesis submitted in conformity with the requirements for the degree of Master of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto

Copyright © 1999 by Kenneth Chalmers
The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author’s permission.

L’auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L’auteur conserve la propriété du droit d’auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-45983-7
Abstract

A CMOS DVD 4x Viterbi Detector: System Design and VLSI Implementation

Kenneth Chalmers
Master of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto
1999

DVD represents the current state-of-the-art in consumer read-only optical storage, and read channel data rates are constantly being pushed higher. At higher data rates, impairments limit the usefulness of traditional slicer based methods of data detection. Also, intersymbol interference degrades the slicer error performance. This thesis examines Viterbi detection, which improves bit error rate, for a 4x (104.64 Mbps) DVD read channel. A model of the DVD read channel is presented, and used as the basis for designing a Viterbi detector that exploits the channel characteristics such as run-length limited coding constraints. An architecture implementing the detector is developed, as well as a 0.35 μm, 3LM, CMOS implementation with a core area of 0.62 mm² and total area of 5.4 mm², which is expected to run at 4x data rates.
Acknowledgements

This work would not have been possible without the support of many people and organizations, and I would like to take this opportunity to thank a few of them here.

First and foremost I would like to thank my parents, whose love and support made this whole incredible journey possible. Thanks Mom; thanks Dad.

Of course this thesis would not have been possible without the inspiration, guidance, assistance, support, and nudging of my supervisor, Professor Glenn Gulak. I am proud to have had the privilege of working under him.

Financial support was provided by NSERC, and the fabrication of the chip would not have been possible without the funding and services of CMC.

Many thanks to the people of LP392 and EECG, past and present, resident and visitor, who have been my companions on this journey. Andy, Christian, Ali, Javad, Sandy, Jordan, Vaughn, Kambiz, Tooraj, Scott, Yaska, Emmanuel, Jason, Marcus, Elias, Qiang, Khalid, and anyone else I’ve forgotten to list. Special thanks to Vincent, Warren, and Nirmal for their friendship and help in getting this project done.

Thanks to Peter, Eugenia, Jaro, and all the other technical people who keep the labs and the tools humming.

I am indebted to Fatih Sarigoz for source code without which the duty-cycle correction code in this thesis would not have been possible.
# Contents

List of Tables vi

List of Figures vii

List of Acronyms x

List of Symbols xii

1 Introduction 1

1.1 Motivation ........................................ 1

1.2 Objectives ........................................ 2

1.3 Outline ........................................ 2

2 Background Theory 4

2.1 Introduction ........................................ 4

2.2 DVD Basics ........................................ 4

2.2.1 A Note on Terminology ........................... 4

2.2.2 Physical Disc Format ............................ 5

2.2.3 The DVD Read Channel .......................... 8

2.3 Data Detection for DVD .......................... 16

2.3.1 Threshold Detection ............................ 17

2.3.2 Viterbi Detection .............................. 18
2.4 Trellis Reduction ........................................... 25
2.5 Existing Implementations .................................. 28
  2.5.1 Hitachi 1997 ............................................. 28
  2.5.2 Kim, Cho, et al. 1998 ................................. 28
  2.5.3 Samsung 1997 ........................................... 28
  2.5.4 Hwang, Lee, et al. 1998 ............................... 28
  2.5.5 Pioneer 1998 ............................................. 29
  2.5.6 Cirrus Logic 1999 ....................................... 29
2.6 Summary ..................................................... 29

3 System Design and Simulation ................................. 31
  3.1 Introduction ............................................... 31
  3.2 Channel Model ............................................ 31
    3.2.1 Impulse Response .................................... 32
    3.2.2 Channel Impairments ................................. 37
    3.2.3 Final Simulation Model ............................... 42
  3.3 Equalization ............................................... 43
  3.4 Simulation Results ....................................... 45
  3.5 Summary .................................................... 48

4 Hardware Design of the Detector .............................. 49
  4.1 Introduction ............................................... 49
  4.2 Design Parameters ......................................... 49
    4.2.1 Quantization .......................................... 50
    4.2.2 Branch Metrics ....................................... 52
    4.2.3 Number of States ..................................... 53
    4.2.4 Merge Depth .......................................... 57
    4.2.5 Survivor Memory Management .......................... 57
List of Tables

2.1 Storage capacity of various read-only optical disc systems. .................. 6
2.2 Attributes of DVD vs. CD. ................................................. 6
2.3 Partial EFMPlus coding table. ........................................... 14
3.1 ZNL simulation parameters .................................................. 40
3.2 DVD specification for the equalizer characteristics. ................. 43
4.1 Default branch metric reference values. ................................. 63
4.2 IC parameters. ................................................................. 69
4.3 IC pin list and functions. .................................................. 71
A.1 Sampling rates. ............................................................... 84
# List of Figures

2.1 Feature sizes of a DVD disc. .................................. 7
2.2 The DVD read channel. ........................................ 8
2.3 NRZI encoding and decoding processes. .................. 9
2.4 Power spectral density of the EFMPlus code (based on figure in [6]). . . 10
2.5 Demonstration of intersymbol interference. .............. 11
2.6 DVD coding and modulation chain chain. .............. 13
2.7 EFMPlus code state transition diagram. ............... 14
2.8 Structure of Reed-Solomon product code block. ........ 16
2.9 Basic conceptual model of threshold detection. ....... 17
2.10 Error event in threshold detection. ...................... 18
2.11 Markov process state transition diagram and its trellis representation. . 21
2.12 Labeled trellis stage. ...................................... 22
2.13 Block diagram of the major operations in the Viterbi algorithm. ...... 23
2.14 Shift register model of intersymbol interference. ...... 25
2.15 $\nu = 3$ trellis diagram (8 states). .................. 26
2.16 Reduced trellis for $\nu = 3$ using $d = 2$ RLL constraint. ...... 27

3.1 Shift-register based channel response model. .......... 32
3.2 Channel impulse response (Gaussian spot profile). ...... 33
3.3 Channel impulse response (least-squares recovery). .... 36
3.4 Impairments in the channel model. ...................... 37
3.5 Zero-memory nonlinearity input/output characteristic. ............... 39
3.6 Block diagram of DVD read channel simulation model. .................. 42
3.7 Digital equalizer characteristic. ........................................... 44
3.8 Sampled DVD data filtered through the channel low-pass filter. ....... 45
3.9 Comparison of channel simulation and sampled DVD readout data. ..... 46
3.10 Error histograms for channel simulation, with and without ZNL. ...... 47
3.11 Sampled DVD data vs. channel simulation noise power spectral density. 47

4.1 Block diagram of the Viterbi detector. .................................... 50
4.2 BER vs. SNR for varying levels of quantization accuracy ($\nu = 4$, jitter $= 10\%$, merge depth $= 25$). .................................................. 51
4.3 BER vs. SNR for L1 and L2 norm branch metrics ($\nu = 4$, quantization $= 5$ bits, jitter $= 10\%$, merge depth $= 25$). ......................... 53
4.4 Impulse responses in terms of channel bit rate. ............................ 54
4.5 BER vs. SNR for varying numbers of states (no quantization, jitter $= 10\%$, merge depth $= 25$). .................................................. 55
4.6 Reduced-state trellis for the DVD read channel. ............................ 56
4.7 Register exchange implementation for 2 state trellis. ..................... 58
4.8 BER vs SNR for final simulation ($\nu = 4$, q=5 bits, merge depth = 25, L1 norm, state metric overflow). ................................. 60
4.9 One block of the branch metric generator. .................................. 61
4.10 The branch metric generator. ............................................... 62
4.11 Comparison value storage. .................................................... 63
4.12 ACS unit building block. ...................................................... 64
4.13 Add-compare-select unit. ..................................................... 65
4.14 Normalization block for ACS. ............................................... 66
4.15 Survivor memory unit slice. .................................................. 67
4.16 The survivor memory unit. ................................................... 67
4.17 Clock doubling and swinging-input design. ............................. 68
4.18 Input/output timing diagram for Viterbi detector chip. ............... 68
4.19 Mask layout for Viterbi detector. ........................................... 70
A.1 Measurement equipment setup. .............................................. 84
A.2 Block diagram of IC read channel, including measurement point. ....... 85
## List of Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACS</td>
<td>Add-Compare-Select</td>
</tr>
<tr>
<td>AWGN</td>
<td>Additive White Gaussian Noise</td>
</tr>
<tr>
<td>BER</td>
<td>Bit Error Rate</td>
</tr>
<tr>
<td>BMG</td>
<td>Branch Metric Generator</td>
</tr>
<tr>
<td>ISI</td>
<td>Intersymbol Interference</td>
</tr>
<tr>
<td>CD</td>
<td>Compact Disc</td>
</tr>
<tr>
<td>CD-ROM</td>
<td>CD Read Only Memory</td>
</tr>
<tr>
<td>CIRC</td>
<td>Cross-Interleave Reed-Solomon Code</td>
</tr>
<tr>
<td>DVD</td>
<td>Digital Video/Versatile Disc</td>
</tr>
<tr>
<td>DVD-ROM</td>
<td>DVD Read Only Memory</td>
</tr>
<tr>
<td>ECC</td>
<td>Error Correction Code</td>
</tr>
<tr>
<td>EFM</td>
<td>Eight-to-Fourteen Modulation</td>
</tr>
<tr>
<td>Gbits</td>
<td>Gigabits ($10^9$ bits = $1.25 \times 10^8$ bytes)</td>
</tr>
<tr>
<td>GB</td>
<td>Gigabyte ($2^{30}$ bytes = $1.073741824 \times 10^9$ bytes)</td>
</tr>
<tr>
<td>GBytes</td>
<td>Gigabytes ($10^9$ bytes)</td>
</tr>
<tr>
<td>Mbits</td>
<td>Megabits ($10^6$ bits = $1.25 \times 10^5$ bytes)</td>
</tr>
<tr>
<td>MB</td>
<td>Megabyte ($2^{20}$ bytes = $1.048576 \times 10^6$ bytes)</td>
</tr>
<tr>
<td>MBytes</td>
<td>Megabytes ($10^6$ bytes)</td>
</tr>
<tr>
<td>RS-PC</td>
<td>Reed-Solomon Product Code</td>
</tr>
<tr>
<td>SMU</td>
<td>Survivor Memory Unit</td>
</tr>
<tr>
<td>SNR</td>
<td>Signal-to-Noise Ratio</td>
</tr>
<tr>
<td>VHDL</td>
<td>Very high speed integrated circuit Hardware Description Language</td>
</tr>
</tbody>
</table>
List of Symbols

a \hspace{1cm} \text{Modulating (channel) bitstream. } a_k \in -1, +1.

\hat{a} \hspace{1cm} \text{Detected version of } a. \text{ Ideally } \hat{a} = a.

g(t) \hspace{1cm} \text{Channel impulse response.}

x_i \hspace{1cm} \text{Trellis state } i.

\xi_{i,j} \hspace{1cm} \text{Transition from state } x_i \text{ to state } x_j.

s_{i,j} \hspace{1cm} \text{Channel symbol corresponding to transition } \xi_{i,j}.

\lambda(\xi_{i,j}, k) \hspace{1cm} \text{Branch metric for transition } \xi_{i,j} \text{ at time } k.

\Gamma_i(k) \hspace{1cm} \text{State metric for state } x_i \text{ at time } k.

\eta_i(k) \hspace{1cm} \text{Decision for survivor path coming into state } x_i \text{ at time } k.

\hat{x}_i(k) \hspace{1cm} \text{Survivor sequence coming into to state } x_i \text{ at time } k.

n(x) \hspace{1cm} \text{Gaussian normal distribution } n(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-\mu)^2}{2\sigma^2}}.

\text{with mean } \mu \text{ and standard deviation } \sigma.

T \hspace{1cm} \text{Bit interval of data from the DVD channel.}

w(t) \hspace{1cm} \text{Least-squares approximation to } g(t).

y \hspace{1cm} \text{Observed channel values (samples).}

\text{y} = a \ast g, \text{ where } \ast \text{ is the convolution operator.}

z(x) \hspace{1cm} \text{Zero memory nonlinearity operation.}
Chapter 1

Introduction

1.1 Motivation

DVD represents the current state of the art in consumer optical storage systems. Known variously as Digital Video Disc, Digital Versatile Disc, or just DVD, the DVD format has now firmly entrenched itself in the homes of consumers the world over. CEMA (the Computer & Electronics Marketing Association) estimates the sales to dealers in the period March 1997 to July 1999 to be 2.8 million units [1] in the United States alone.

On the data side, DVD-ROM drives are widely available as upgrades to existing computers, and many new computers ship with DVD-ROM drives as standard equipment instead of CD-ROM drives. Dataquest estimates more than 8 million DVD-ROM drives will be shipped in 1999 [2]. As in any computer storage mechanism, research immediately began to increase the data rates of the drives.

Like CD-ROM, the reference multiple ("x") method of speed notation is used, with 1x denoting a first-generation, reference speed DVD drive (channel bit rate of 26.16 Mbps). 2x and 4x drives rapidly appeared, and as of this writing 6x drives are available from some manufacturers (such as Pioneer [3] and Toshiba [4]).

To enable these high data rates, high speed analog and digital electronics are re-
quired to retrieve the data encoded on the disk and return it to the original user data bitstream. While basic threshold detection has served the first generation of DVD players, impairments to the read process (which become magnified at higher speeds) result in increased detector accuracy requirements in the read channel. One such improved method, examined in this thesis, is Viterbi detection.

Also, future high-density DVD (HD-DVD) systems using a blue laser [5] will require Viterbi detection, since increased data density will magnify channel impairments. These channels will also run at higher data rates than first-generation DVD and thus require high speed detection. This thesis will focus solely on standard DVD-Video/ROM discs at high data rates; however, the principles presented here could easily be used to develop a Viterbi detector for HD-DVD when it becomes available.

1.2 Objectives

This thesis describes the development of a DVD read channel detector. We present the design of a channel simulation for evaluating performance of various schemes, followed by the system design and VLSI implementation of a Viterbi detector suitable for use in a 4x DVD read channel.

1.3 Outline

The thesis is presented in five chapters and three appendices. Chapter 2 provides the background theory used in this thesis. It describes the DVD system and its read channel, data detection systems used in the DVD read channel (including the Viterbi algorithm), the tuning of the Viterbi algorithm to run-length limited channels, and existing implementations. Chapter 3 presents the design of a DVD read channel system simulation, including channel model, equalization, and comparisons of our model to actual DVD read channel data obtained with an oscilloscope. Chapter 4 describes the hardware im-
implementation of the Viterbi detector, including tradeoffs made in adapting the algorithm to hardware, a description of the architecture used, and a summary of the IC implementation. Finally, Chapter 5 summarizes the work and contributions of this thesis and presents suggestions for future improvements and research. Appendix A describes the experimental measurement setup used to obtain readout signals from an existing DVD player, Appendix B contains the C source code for our channel model, and Appendix C contains the VHDL source code for the Viterbi detector implementation.
Chapter 2

Background Theory

2.1 Introduction

In this chapter we provide some background theory on the systems and algorithms used in this thesis. First, a brief introduction to the DVD system, including physical disc format and read channel basics, is presented. Next, we discuss the detection schemes used in the DVD read channel: threshold detection and Viterbi detection. A method of reducing the Viterbi trellis to fit the run-length limiting constraints of the channel follows. Finally, we conclude the chapter with a summary of existing literature on detection in the DVD read channel.

2.2 DVD Basics

2.2.1 A Note on Terminology

Since its introduction, there has been much confusion about the acronym DVD. Originally it stood for “Digital Video Disc,” since DVD was first conceived as a video counterpart to the CD (Compact Disc) of the audio world. However, early on it was decided that the DVD format would be useful for a wider range of applications than just dis-
playing movies, such as audio (entire libraries of a composer’s work could fit on a single DVD) and computer data (DVD-ROM, a direct replacement for CD-ROM and backwards compatible in that any DVD-ROM drive can read CD-ROM discs). Thus, it was proposed that the acronym (which by now had been too widely adopted to change) have its meaning altered to "Digital Versatile Disc." After some argument within the DVD community, the general consensus seems to be to use the acronym DVD without committing it to stand for any particular meaning. This thesis will follow the latter convention by referring to the format throughout simply as DVD.

2.2.2 Physical Disc Format

DVD was designed as a successor to the ubiquitous Compact Disc (CD) and CD-ROM. A DVD disc shares a CD’s macroscopic dimensions. Visibly a DVD is identical to a CD, with a 120 mm diameter and 1.2 mm thickness.

Like other read-only optical storage mechanisms, DVD stores information as runs of pits and lands (low levels and high levels) etched into the substrate of the physical disc. A laser shines on the disc and reflects off the surface back to a detector. The intensity sensed at the detector varies depending on whether the laser is shining on a pit or a land. These intensity variations are translated through some signal processing algorithms back into the original user data stream that was written to the disc.

The primary advantage of DVD over CD is increased storage capacity. A CD-ROM has a storage capacity of 680 Mbytes of data, while a single-sided single-layer DVD can hold 4.7 Gbytes. Furthermore, DVDs can be manufactured with two separate layers of information stacked vertically on one side of the disc (which can be accessed without flipping the disc). The upper layer is semi-transparent, allowing the laser to focus through it to the second layer below. Finally, DVDs can also be manufactured with information on both sides of the disc (which requires disc flipping). The formats are usually abbreviated SL/DL for single layer/double layer and SS/DS for single-sided/double-sided. Table 2.1
Table 2.1: Storage capacity of various read-only optical disc systems.

<table>
<thead>
<tr>
<th>Disc Format</th>
<th>Capacity</th>
</tr>
</thead>
<tbody>
<tr>
<td>CD</td>
<td>680 Mbytes</td>
</tr>
<tr>
<td>DVD (SSSL)</td>
<td>4.7 GBytes</td>
</tr>
<tr>
<td>DVD (SSDL)</td>
<td>8.54 GBytes</td>
</tr>
<tr>
<td>DVD (DSSL)</td>
<td>9.4 GBytes</td>
</tr>
<tr>
<td>DVD (DSDL)</td>
<td>17.08 GBytes</td>
</tr>
</tbody>
</table>

Table 2.2: Attributes of DVD vs. CD.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>CD</th>
<th>DVD</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Track pitch</td>
<td>1.6 μm</td>
<td>0.74 μm</td>
<td>54%</td>
</tr>
<tr>
<td>Minimum pit/land length</td>
<td>0.834 μm</td>
<td>0.40 μm</td>
<td>52%</td>
</tr>
<tr>
<td>Substrate thickness</td>
<td>1.2 mm</td>
<td>0.6 mm × 2</td>
<td>0%</td>
</tr>
<tr>
<td>Recording code (rate)</td>
<td>EFM (8/17)</td>
<td>EFM Plus (8/16)</td>
<td>Rate 6.25%</td>
</tr>
<tr>
<td>Error control coding</td>
<td>CIRC</td>
<td>RS-PC</td>
<td></td>
</tr>
<tr>
<td>Error control coding rate</td>
<td>[32,28]/[28,24]</td>
<td>[208,192]/[182,172]</td>
<td>16.3%</td>
</tr>
<tr>
<td>Laser wavelength</td>
<td>780 nm</td>
<td>650 nm</td>
<td>17%</td>
</tr>
<tr>
<td>Lens numerical aperture</td>
<td>0.45</td>
<td>0.6</td>
<td>33%</td>
</tr>
<tr>
<td>Reference channel bit rate</td>
<td>4.32 Mbits/s</td>
<td>26.16 Mbits/s</td>
<td>506%</td>
</tr>
<tr>
<td>Reference user bit rate</td>
<td>1.41 Mbits/s</td>
<td>11.08 Mbits/s</td>
<td>686%</td>
</tr>
<tr>
<td>Total storage capacity</td>
<td>0.68 Gbytes</td>
<td>4.7 Gbytes</td>
<td>590%</td>
</tr>
</tbody>
</table>

summarizes the storage capacities of the different options.

As Table 2.1 shows, the double sided, double layer DVD can hold the equivalent of more than 25 CD-ROM discs. Even the basic 4.7 Gbyte DVD translates to a 6.9 times increase in storage capacity over CD. This increase comes about due to improvements in several areas, such as physical geometry, modulation schemes, and coding schemes. The differences between these attributes of CD and DVD are summarized in Table 2.2 [6].

The feature sizes (track pitch and minimum pit/land length) of DVD are much smaller than those of CD - each less than half the size. This allows more data to be packed into a given unit area (increased areal density). Fig. 2.1 illustrates the feature sizes graphically.
Figure 2.1: Feature sizes of a DVD disc.
2.2.3 The DVD Read Channel

This thesis is concerned with the detection of data from DVD discs, converting the analog waveforms received from the laser back into the digital bitstream that was originally encoded on the disc. To provide context for the detection process, this section will describe the major building blocks of the DVD read channel. Figure 2.2 shows these basic blocks.

![Figure 2.2: The DVD read channel.](image)

Channel Detector

The channel detector converts the analog readout signal from the optical pickup into the original channel bitstream. This block is the focus of this thesis, and will be explored further in section 2.3.

NRZI

NRZI (Non return to zero inverted) is a common modulation scheme in communication systems. It encodes the data as transitions rather than levels. During a bit interval, the presence of a high-to-low or low-to-high transition signifies a 1, while the absence of such a transition represents a 0. For example, if the low signal is represented by 0 and the high
signal by 1, then the bit stream 00101 would be represented (assuming an initial signal level of 0) as 0, 0, 1, 1, 0.... Using Boolean algebra, this procedure can be described as exclusive-oring (XOR) the previous bit with the current bit (Eq. 2.1).

\[ y_k = x_k \oplus y_{k-1}, \quad y_{-1} = 0 \]  

(2.1)

A property of NRZI, due to its XOR nature, is that recovering the NRZI modulated bits simply involves performing the exact same XOR operation on the modulated bitstream. Since:

\[ (a \oplus b) \oplus b = a \oplus (b \oplus b) = a \oplus 0 = a \]  

(2.2)

the original bitstream can be recovered using:

\[ x_k = y_k \oplus y_{k-1}, \quad y_{-1} = 0 \]  

(2.3)

Figure 2.3 shows the encoding and decoding processes.

Figure 2.3: NRZI encoding and decoding processes.

In the DVD write process, the NRZI modulated bits are converted from 1/0 to +1/-1 before writing, so the signal being detected follows the +1/-1 convention. Once recovered, the NRZI demodulated bits are fed into the EFMPlus demodulator, described next.
EFMPlus

EFMPlus coding is slightly more complicated than NRZI. Though the acronym EFM for “Eight-to-fourteen modulation,” this is slightly misleading since the scheme encodes 8 user bits into 16 channel bits (a rate 8/16 code). Still, EFMPlus is an improvement over the older EFM scheme used in CD-ROM, which is a rate 8/17 code.

The purpose of EFMPlus coding is twofold: DC-suppression and run-length limiting. The EFMPlus recording code was carefully designed to minimize the DC content of a random input bitstream. This feature ensures that the channel data stream doesn’t interfere with low frequency signals used in the servo system (which controls the alignment of the optical pickup). Immink ([6], [7]) and Braun and Janssen [8] discuss the DC suppression of EFMPlus. Figure 2.4, adapted from [6], shows the computer simulated power spectral density of EFMPlus modulated data (the lower curve represents the code performance with 3 bytes of lookahead, which improves DC suppression).

![Power spectral density of EFMPlus code](image)

Figure 2.4: Power spectral density of the EFMPlus code (based on figure in [6]).
Run-length limiting (RLL) [9] is the other major function performed by EFMPlus. RLL codes are widely used in computer disc drives to mitigate the effects of intersymbol interference (ISI).

ISI is a result of the isolated impulse response of a channel spanning two or more bit intervals. In other words, some of the energy from previous and/or future symbols interferes with the energy of the current symbol at the sampling instant, either constructively or destructively. Figure 2.5 demonstrates ISI graphically.

![Figure 2.5: Demonstration of intersymbol interference.](image)

Ideally, a channel has zero ISI. This states that, in the digital domain, the impulse response $p$ must satisfy ([10] chapter 6):

$$p(kT) = \delta_k$$

or its Fourier transform must satisfy:

$$\frac{1}{T} \sum_{m=-\infty}^{\infty} P \left( j\omega - jm\frac{2\pi}{T} \right) = 1$$
This is the Nyquist criterion for zero ISI.

Unfortunately, many real-world channels are band limited and the bandwidth available or spectral shape of the channel response does not allow impulse responses which have zero ISI. The DVD channel falls into this category and so the ISI must be accommodated, through the RLL feature of EFMPlus.

RLL codes help to mitigate ISI by imposing constraints on the output data stream. They require that a certain number of ‘0’ or null symbols be present between any consecutive ‘1’ symbols in the original data stream. The number of required zeros is the $d$ parameter of an RLL code. Also, RLL codes often impose an upper limit on the number of consecutive nulls, denoted $k$, which helps to ensure adequate information is available for timing recovery (e.g. PLL locking). DVD’s EFMPlus code has $(d, k) = (2, 10)$ [6]. This means that, after NRZI modulation, any bit interval with a transition in it (denoting a ‘1’) will be followed by at least 3 bit intervals ($3T$) with no transition.

A constraint of $d = 2$ translates to a $3T$ run length due to the nature of NRZI. For example, with an initial condition of 0, the bit sequence 1001 would be output as the NRZI modulated sequence 1, 1, 1, 0 (following Eq. 2.1). The two zeros result in three consecutive ones.

Figure 2.6 shows the entire coding and modulation process (including NRZI) graphically.

The EFMPlus coding procedure is implemented using a 4-state finite state machine with a lookup table in each state. Each lookup table has 344 16-bit words in it and a next state specification. The EFMPlus modulator takes streams of 8-bit input words and uses the values as indices into the table. From that table position, the state machine reads the 16-bit output word and the next state. A partial coding table for the EFMPlus scheme is shown in Table 2.3 (from [6]). Figure 2.7 shows the state transition diagram, where each state is labeled with the transition set $S_{xy}$ that represents the set of all input values that cause a transition from state $x$ to state $y$. For example, looking at Table 2.3
only inputs 6 and 8 have a next state of 3 from a current state of 2, so they would be in the set $S_{23}$.

Note that an 8-bit input word implies 256 lookup positions, while the tables have 344 symbols available. The extra 88 words make up the substitution table, and their values correspond to the first 88 input words (0000 0000 through 0101 0111). These substitution table values, along with lookahead during the coding process, are used to improve DC suppression. By considering the next 2–3 values that will be generated, the modulator can choose the output value, either from the main or substitute table, which will minimize the DC content.

The decoding process involves examining the bitstream in 16-bit words (synchronized so they correspond to the modulator output through the use of a reserved sync word) and using an inverse lookup table to recover the original 8-bit input. Some of the 8-bit input words are represented by the same 16-bit output word (for example inputs 5 and 6 in state 1); however, these were carefully designed so that though the outputs are the same the next states are always either state 2 or state 3, and never the same. States 2

Figure 2.6: DVD coding and modulation chain chain.
### Table 2.3: Partial EFMPlus coding table.

<table>
<thead>
<tr>
<th>Input</th>
<th>State 1 (out, next)</th>
<th>State 2 (out, next)</th>
<th>State 3 (out, next)</th>
<th>State 4 (out, next)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>001000000001001, 1</td>
<td>010000100100000, 2</td>
<td>001000000001001, 1</td>
<td>010000100100000, 2</td>
</tr>
<tr>
<td>1</td>
<td>001000000001001, 1</td>
<td>001000000001001, 1</td>
<td>100001001000000, 3</td>
<td>100000100100000, 3</td>
</tr>
<tr>
<td>2</td>
<td>0010000010010000, 2</td>
<td>0010000010010000, 2</td>
<td>100000001001000, 1</td>
<td>100000001001000, 1</td>
</tr>
<tr>
<td>3</td>
<td>00100000001001000, 2</td>
<td>010001001000000, 4</td>
<td>0100000001001000, 2</td>
<td>010001001000000, 4</td>
</tr>
<tr>
<td>4</td>
<td>0010000010010000, 2</td>
<td>0100000100100000, 2</td>
<td>100001001000000, 2</td>
<td>100000100100000, 2</td>
</tr>
<tr>
<td>5</td>
<td>0010000000100100, 2</td>
<td>0010000000100100, 2</td>
<td>1001001000000000, 4</td>
<td>1001001000000000, 4</td>
</tr>
<tr>
<td>6</td>
<td>0010000000100100, 3</td>
<td>0010000000100100, 3</td>
<td>1001001000000000, 4</td>
<td>1001001000000000, 4</td>
</tr>
<tr>
<td>7</td>
<td>0010000001001000, 3</td>
<td>0100000000100100, 1</td>
<td>0010000001001000, 3</td>
<td>0100000000100100, 1</td>
</tr>
<tr>
<td>8</td>
<td>0010000010010000, 3</td>
<td>0010000010010000, 3</td>
<td>1000100100000000, 4</td>
<td>1000100100000000, 4</td>
</tr>
</tbody>
</table>

![EFMPlus code state transition diagram](image)

Figure 2.7: EFMPlus code state transition diagram.
and 3 were assembled so that bits 1 (the MSB) and 13 unambiguously identify the state (output words taken from state 2 always have zeros in both these bits, while those from state 3 have at least one one). Thus, with the knowledge of these two bits from the next modulated word the current word can be unambiguously demodulated.

After EFMPlus decoding the channel bits are now ready for error control code decoding, discussed next.

**Error Control Code (ECC)**

After demodulation, one more step is required to transform the channel bits back into the original user bits: error control code decoding. Error control coding uses parity bits stored with the original user bits to detect and correct errors in the data stream it receives. The DVD channel uses a scheme known as Reed-Solomon product code (RS-PC) for its error control coding.

RS-PC views the data as matrices of bytes that are input from the EFMPlus demodulated bitstream, 192 rows by 172 columns. It uses two Reed-Solomon codes, $C_1$ and $C_2$, to generate parity checks for the data. $C_1$ is a $[208,192]$ RS code, meaning that for every 192 byte long input string, it generates a 208 byte long output string (i.e. there are $208 - 192 = 16$ parity bytes). $C_2$ is a $[182,172]$ code.

To generate an RS-PC block requires two steps:

- Apply $C_1$ to the columns of the $192 \times 172$ input matrix to generate a $208 \times 172$ matrix.

- Apply $C_2$ to the rows of the $208 \times 172$ matrix to generate a final $208 \times 182$ byte output matrix.

Note that $C_2$ operates on every row, including those containing only parity bytes generated by $C_1$. This additional redundant error check improves the performance of the scheme. Figure 2.8 shows the structure of an RS-PC block.
The RS-PC ECC code is capable of correcting a maximum burst error length of approximately 2200 bytes (5.8% of an ECC block, 4.6 mm worth of data on the physical disc) and can reduce a random input error rate of $2 \times 10^{-2}$ to user data error rate $10^{-15}$ [6].


Source Decoding

Finally, the fully detected and decoded user bits are demultiplexed into their component streams (audio, video, subtitles, and others) and sent to the various source decoders required to convert the bitstreams into audio signals (such as Dolby Digital [13]), video signals (MPEG-2 video [14]), and others. These source decoders are beyond the scope of this thesis.

### 2.3 Data Detection for DVD

Data detection for the DVD read channel is the focus of this thesis. This section will briefly describe threshold detection, the traditional method of detection used in CD
players and first generation DVD drives. We will then describe Viterbi detection, the application of the Viterbi algorithm to data detection, which improves detector error rates in the DVD read channel.

2.3.1 Threshold Detection

Threshold detection is the simplest method of data detection, and has proved effective in multiple generations of CD-ROM drives and the first generation of DVD drives.

Conceptually, threshold detection involves slicing the input signal to a high or low level and sampling the sliced signal at a rate synchronized to the rate of the data symbols. Figure 2.9 shows this basic model.

![Threshold Detection Diagram](image)

Figure 2.9: Basic conceptual model of threshold detection.

In an ideal channel, this scheme is perfectly adequate. However, as impairments start distorting the channel response, the number of errors made by a threshold detector can rapidly become unacceptable. Because the output of the detector is highly dependent on
the magnitude of the input signal, any impairment (such as ISI, random noise, or timing jitter) that can reduce the magnitude of the signal at the sampling instants increases the probability of making an error (the slicer output being inverted). Figure 2.10 shows an example of an error event in a threshold detector system due to timing jitter.

![Figure 2.10: Error event in threshold detection.](image)

In a channel with intersymbol interference, such as the DVD read channel under consideration, the ISI causes signal energy loss at the sampling instants due to deconstructive interference. Instead of binary signaling, the channel symbols take on multiple values (up to $2^l$, where $l$ is the length of the channel impulse response in bit intervals).

This generally results in a lower average signal energy being presented to the slicer, which reduces its noise immunity (lowers the SNR). However, by examining adjacent bits (something the basic slicer cannot do) we can account for the different signal levels (channel symbols). A detector that takes the ISI into account and uses the information presented by it could improve error performance. One such detector, using the Viterbi algorithm, is discussed next.

### 2.3.2 Viterbi Detection

Viterbi detection is based on the Viterbi algorithm ([15], [16]), which is a dynamic programming algorithm for finding the shortest path through a class of directed, weighted
graphs. It can be shown that, in the presence of additive white Gaussian noise (AWGN), the Viterbi algorithm falls into the class of maximum-likelihood sequence detectors (MLSD). These detectors are optimal in the maximum-likelihood sense, that is they maximize the probability density function:

\[ f_{Y|S}(y|s) \]  

where \( y \) is the observed channel value (corrupted by noise) and \( s \) is a channel symbol (the set of all channel symbols being denoted \( S \)). This is the likelihood function for the symbol \( s \).

In the presence of AWGN, the likelihood function becomes ([10] chapter 9):

\[ f_{Y|S}(y|s) = \frac{1}{2\pi\sigma^2} e^{-\frac{(y-s)^2}{2\sigma^2}} \]  

(2.7)

where \( \sigma^2 \) is the noise variance.

Since the likelihood function is nonzero, we can take the natural logarithm and negate each side of the identity without affecting it. This means Eq. 2.7 can be rewritten as:

\[-\ln[f_{Y|S}(y|s)] = \ln(2\pi\sigma^2) + \frac{(y-s)^2}{2\sigma^2} \]  

(2.8)

Since \( \sigma \) is a constant, maximizing the likelihood function in AWGN is equivalent to finding the symbol \( s \) which minimizes:

\[ (y-s)^2 \]  

in Eq. 2.8.

The maximum-likelihood sequence detector maximizes the likelihood (minimizes Eq. 2.9) over the entire sequence of received values \( y \), choosing the set of symbols \( s \) so that Eq. 2.9 is minimized for the global sum of all likelihoods (possibly at the expense of some individual components). That is, it finds \( s \) so that:

\[ \sum_{k=1}^{K} (y_k - s_k)^2 \]  

(2.10)
is minimized (where $K$ is the message length).

This vector interpretation shows that maximum likelihood detection is equivalent to finding the vector $s$ that is closest to $y$ in Euclidean space—it minimizes the distance between the two vectors. Thus the maximum likelihood detector in the presence of AWGN is also known as a minimum distance detector. The Viterbi algorithm can be used as a minimum distance/maximum likelihood sequence detector.

We will present a quick overview of the Viterbi algorithm in this section. First, we describe the trellis diagram representation of a finite-state discrete time Markov process. This is a directed weighted graph suitable for use with the Viterbi algorithm. We will then provide a brief description of the algorithm itself as used in the detector. In conclusion, we show how an ISI channel such as the DVD read channel can be represented using a trellis diagram and therefore utilize the Viterbi algorithm.

**Trellis diagrams**

Introduced in [17], a trellis diagram is a representation of a finite-state discrete-time Markov process with the transitions unrolled in time. Each stage in the trellis shows every state in the process and all possible transitions leading into and out of that state. Figure 2.11 shows the familiar state-transition diagram description for a 2-state Markov process and the corresponding trellis for 5 time intervals.

We denote the set of all possible states of the process as $X$, with $x_i$ being state $i$ at some arbitrary time. There are $M$ total states (2 in Fig. 2.11), so $i$ can take on the values $0 \leq i \leq M - 1$. The transition from a current state $i$ to a future state $j$ from the next time slice is defined as $\xi_{i,j}$. These are the edges in the trellis diagram.

The edges of the trellis are weighted with lengths, commonly known as branch metrics and denoted $\lambda(\xi_{i,j}, k)$. It is the sum of these lengths between the starting and ending nodes in the trellis that the Viterbi algorithm aims to minimize, thus finding the shortest path through the trellis.
Each state at a given time $k$ also has a state metric associated with it, $\Gamma_i(k)$, which is the total length of the shortest path leading up to state $i$ at time $k$.

Associated with a given $\Gamma_i(k)$ are the output values associated with the branches that resulted in that state metric. These outputs concatenated together form the survivor sequence leading up to state $i$, which we call $\hat{x}_i(k)$.

Figure 2.12 shows one stage of the 2-state trellis labeled with this notation.

With this notation defined, we can now describe the Viterbi algorithm solution to finding the shortest path through a trellis.

The Viterbi algorithm

Given a trellis such as that of Fig. 2.11, and a known starting state $x_0$, the Viterbi algorithm operates as follows (from [16]):

1. Initialize storage:
   
   - $k = 0$
Figure 2.12: Labeled trellis stage.

- $\hat{x}_s(0) = x_s$; $\hat{x}_i(0)$ ($i = 1...M$, $i \neq s$) arbitrary.
- $\Gamma_s(0) = 0$; $\Gamma_i(0)$ ($i = 1...M$, $i \neq s$) = $\infty$

2. Compute:

$$\Gamma(\xi_{i,j}) \triangleq \Gamma_i(k) + \lambda(\xi_{i,j}, k)$$

for all $\xi_{i,j}$.

3. For each $x_j$ at time $k + 1$ find:

$$\Gamma_j(k + 1) = \min_i \Gamma(\xi_{i,j})$$

and store $\Gamma_j(k + 1)$ and the corresponding survivor sequence $\hat{x}_j(k + 1)$.

4. Let $k = k + 1$ and repeat at step 2 until the entire trellis is traversed.

The fundamental operations of the algorithm can be represented with three major blocks, as shown in Fig. 2.13.
Figure 2.13: Block diagram of the major operations in the Viterbi algorithm.

In the literature, the blocks for branch metric calculation, path selection, and survivor sequence storage are known as the branch metric generator (BMG), add-compare-select unit (ACS or ACSU), and survivor management unit (SMU) respectively. An architecture for hardware implementation of the Viterbi detector will be examined in section 4.3.

**Branch metric calculation**

For channel detection, the common method used for branch metric calculation is the Euclidean distance squared (L2 norm) between the observed signal at time $k$ ($y_k$) and a set of $I$ (possibly time varying) possible channel symbols $s_{i,j}(k)$, $1 \leq i \leq I$. In the detector studied here the channel symbols will be constant with time, so we drop the function-of-$k$ notation and label them $s_{i,j}$. With this we can calculate the branch metrics at each time step as:

$$\lambda(\xi_{i,j}, k) = (y_k - s_{i,j})^2 \text{ for all transitions } \xi_{i,j}$$  \hspace{1cm} (2.11)

The branch metric with the smallest L2 norm is the closest in Euclidean space to the actual received signal. The Viterbi algorithm minimizes the sum of the L2 norms over the signal sequence, thus minimizing the distance between the vector of the received inputs ($y$) and the set of all vectors of possible input sequences $s$ which, in the presence of AWGN, corresponds to maximum-likelihood detection.
Trellis description of a channel with ISI

The Viterbi detector can be used in a channel with intersymbol interference [16]. The following is a brief summary of how this is achieved.

The digital model of a communications channel, given digital impulse response \( g \) and input stream \( a \), yields an output \( y \) such that:

\[
y_k = \sum_{m=-\infty}^{\infty} a_m g_{k-m}, \forall k
\]

(2.12)

In a system with no ISI \( g_k \) is an impulse, resulting in \( y_k = a_k \). In the DVD channel, \( a_k \) takes on the values +1 or -1 representing a bit 1 or bit 0 respectively.

Commonly \( g_k \) is not an isolated impulse but still has finite energy. This means the amount of ISI (width of the impulse response) decays to zero or becomes negligible after a certain time span \( \nu \), meaning \( g \) is finite with extent \( \nu \). This reduces Eq. 2.12 to:

\[
y_k = \sum_{m=k-\nu+1}^{k} a_m g_{k-m}
\]

(2.13)

Eq. 2.13 describes a shift register process of length \( \nu \), where the values tapped from the shift register are multiplied by the samples of the impulse response. Figure 2.14 shows the shift register digital model of ISI. The non-causal response shown becomes causal in the model - this transformation doesn't matter in our application since all simulations are run off-line (non real-time), so timing delays that are integer multiples of the symbol rate make no difference to the system.

A shift register sequence can be modeled as a finite-state discrete-time Markov process, which is exactly what a trellis describes. Thus, we can map the ISI model into a trellis diagram.

The length of the shift register describing the channel impulse response is also known as the constraint length. The resulting number of states in the trellis is \( 2^\nu \).

Each state in the trellis represents one possible sequence of received symbols. The transitions represent the results of shifting out the oldest symbol in the state and shifting...
in a new symbol, either 0 or 1 in our case. Thus, \( s_{i,j} \) for each transition \( \xi_{i,j} \) is the output from the filter after setting the bits in the shift register equal to the string of bits for that state and transition.

### 2.4 Trellis Reduction

The previous section introduced the concept of a trellis diagram, a time unrolled representation of a Markov process. In Section 2.2.3 we also introduced RLL codes, a modulation technique used to combat intersymbol interference. In this section we will present a method for exploiting the \( d \) constraints of RLL codes to reduce the number of states in the trellis for a channel.

Figure 2.15 shows a standard trellis for a binary channel with constraint length \( \nu = 3 \) (8 states). The dashed lines correspond to receiving the symbol zero, while the solid lines represent receiving the symbol one.

In the figure, each state is labeled with a binary number representing the last \( \nu \) bits received (3 in this example). Transitions are labeled with either a '0' or a '1', representing a transition based on receiving the corresponding bit. Each state label is unique, and represents one of the 8 different values that can be taken on by a 3 bit binary number.
Figure 2.15: $\nu = 3$ trellis diagram (8 states).
Not all these numbers are valid according to our RLL constraints. For example, a 11 before NRZI (which violates our \( d = 2 \) constraint) becomes either 010 or 101 (depending on whether the previous bit is a 1 or 0) after NRZI modulation. Thus states 2 and 5 are not legal according to our modulation scheme and can be removed from the trellis diagram altogether.

Using the same principle, certain transitions can be removed as well. For example, state 1 (binary 001) followed by a 0 (giving the binary sequence 0010) would be NRZI demodulated as either 0011 or 1011, both of which violate the \( d \) constraint. Removing all the other transitions with violations, we arrive at the trellis shown in Fig. 2.16.

![Trellis diagram](image)

Figure 2.16: Reduced trellis for \( \nu = 3 \) using \( d = 2 \) RLL constraint.

The benefits of trellis reduction are twofold. First, this improves detection accuracy by not allowing the detector to even consider the impossible states. Second, the hardware architecture of the detector is simplified, since each state corresponds to a number of hardware resources in the implementation.
2.5 **Existing Implementations**

In this section we will review the known existing publications (to September 1999) involving DVD read channel detectors.

### 2.5.1 Hitachi 1997

A paper by research teams with Hitachi Ltd. describes a 2x DVD-ROM read channel implementation in [18]. Their implementation focuses on the equalizer design with a brief discussion of the ECC decoder. There is no discussion of the detection method used or BER performance achieved.

### 2.5.2 Kim, Cho, et al. 1998

In [19] the development of the analog front-end for a 4x DVD read channel IC is given. This front-end uses threshold detection, and focuses on the analog signals (pick up tracking, equalizer, etc.) rather than the digital sections. No detector error performance curves are shown.

### 2.5.3 Samsung 1997

An implementation by the Korean electronics company Samsung is given in [20]. This implementation uses a partial response transfer function in its design, meaning their implementation requires a partial response equalizer front-end. They describe their research, but present no hardware implementation.

### 2.5.4 Hwang, Lee, et al. 1998

This research, described in [21], introduces the zero-memory nonlinearity (ZNL) as an impairment in the DVD read channel. We will examine the ZNL impairment further
in Section 3.2.2. The paper describes an adaptive Viterbi detector using a least-mean-squares approach to estimate the parameters of the ZNL and update the branch metric generation accordingly. This includes only computer simulation results, no hardware implementation.

2.5.5 Pioneer 1998

Pioneer, the Japanese electronics company, describes a Viterbi detector for DVD read channel in [22]. They state that their design has been incorporated into a read channel LSI chip, but provide no operating details or design parameters.

2.5.6 Cirrus Logic 1999

A DVD read channel design operating at 120 million samples per second (4.5x) is presented in [23] by a team from Cirrus Logic Semiconductors. Their paper describes a complete read channel implementation, including servo control, timing recovery, demodulation, and ECC. They briefly mention that the circuit uses a Viterbi detector, but again provide no implementation details. They claim a gain of 6 dB at a bit error rate of $10^{-6}$ using maximum-likelihood sequence detection over slicer-based detection. Their 0.35 μm CMOS implementation is 73 mm$^2$ for the entire mixed-signal read channel.

2.6 Summary

In this chapter we:

- Gave an overview of the DVD system, including improvements compared to its predecessor CD.

- Presented the basic DVD read channel flow, including detection, modulation, and error-control coding, with a brief discussion of the schemes used.
• Discussed detection in the DVD channel in detail, including threshold and Viterbi detection.

• Showed a technique for exploiting run-length limiting constraints to reduce the number of states in a trellis.

• Reviewed existing literature/implementations on Viterbi detection for DVD channels.
Chapter 3

System Design and Simulation

3.1 Introduction

This chapter presents the design and simulation of a DVD read channel. We discuss the development of a channel model, including the impulse response of the linear channel and the impairments (noise) found in the channel. We then describe the equalization used in the DVD read channel, and conclude with a comparison of the channel simulation to data sampled from an actual DVD player channel.

3.2 Channel Model

The channel model is the most important preliminary in the design of the detector. The channel model allows us to experiment with different parameters in the design and see how changing them will affect the performance of the detector. No model is perfect, and a model will never completely reflect the true performance of the detector in the actual physical system. However, we can get an approximation that can demonstrate the relative performance of different schemes, and obtain results that will be close to the absolute results of the physical circuit if the model is accurate. This section will develop a channel model for the DVD read channel used in the design of the Viterbi detector.
3.2.1 Impulse Response

Perhaps the most important aspect of any linear communication system model is the impulse response of the channel. This is the output of the channel in response to a single, isolated impulse at its input. For a linear channel, the response of the channel to a stream of input data can be modeled as the convolution of that data with the channel’s impulse response. In the discrete-time channel with finite impulse response, this can further be simplified to a shift register process such as that of Fig. 3.1.

![Shift-register based channel response model.](image)

The impulse response can be either time-invariant (for static channels) or time-varying for time-varying channels such as fading channels. This thesis only considers a static channel model.

We next examine two impulse response models used over the course of this study: the Gaussian impulse response and the response recovered using least-squares channel identification.

**Gaussian impulse response**

As a first approximation, the DVD channel can be modeled using a simple Gaussian impulse response $h(t)$, as described by Eq. 3.1.

$$h(t) = \frac{2}{t_0 \sqrt{\pi}} e^{-\left(\frac{\pi t}{t_0}\right)^2}$$  \hspace{1cm} (3.1)
This is the approximate impulse response for an optical channel given in [24]. This response is characterized by $t_0$, which controls the 1/e-width of the (Gaussian-shaped) laser spot. For our DVD model, we determined, by using the impulse in simulation and comparing the results to the sampled DVD data, that 150 ns (which is approximately $4T$ at a 1x bit interval of $T = \frac{1}{26.16 \times 10^6}$) was a reasonable value for $t_0$. The resulting impulse response is shown in Fig. 3.2.

![Figure 3.2: Channel impulse response (Gaussian spot profile).](image)

This impulse response was used in all simulations up to and including the design of the integrated circuit, described in Chapter 4. However, further research revealed a better method for determining the impulse response, discussed next.

**Least-squares channel identification**

Least-squares channel identification uses a least-squares (LS) solution technique to extract the impulse response of a linear channel from a readout of the channel's response to a known data sequence.
The least-squares algorithm is a method of solving an overdetermined set of linear equations, i.e. a system with more equations than unknowns. For a linear equation, this represents the before/after channel response bits (the equations) and the channel impulse response (the unknowns).

The full description including performance notes is discussed in [25]. A brief summary of the algorithm follows here:

Let:

- $T$ denote the channel bit interval.
- $y$ denote the read-head output signal.
- $p$ denote the readout oversampling factor.
- $a$ denote the original user bit stream ($a \in +1, -1$ for DVD).
- $g$ denote the channel impulse response.
- $w$ denote the recovered version of $g$.

Then:

$$y(t) = \sum_k a_k g(t - kT) + u(t)$$

where $u(t) = \text{uncorrelated, additive, zero-mean noise}$.

Let our sampling factor be $T_d = \frac{T}{p}$.

Sampling with $t = mT_d$ ($m \in \text{integers}$):

$$y(mT_d) = \sum_k a_k g(mT_d - kT) + u(mT_d)$$

The estimated channel response is:

$$\hat{y}(mT_d) \triangleq \sum_k a_k w(mT_d - kT)$$
where \( w(t) \) is the estimated version of \( g(t) \).

Our problem is to find \( w(t) \) such that the error signal,

\[
\epsilon(mT_d) \triangleq d(mT_d) - \hat{d}(mT_d)
\]

is minimized.

Now define:

- \( M \) is the estimated length of the channel impulse response.
- \( l \) is the number of samples of \( y \) available.

\[
W_{M,l} \triangleq \begin{bmatrix}
  w_l(0) \\
  \vdots \\
  w_l((M-1)T_d)
\end{bmatrix}
\]

\[
A_{M,m} \triangleq \begin{bmatrix}
  a_m \\
  \vdots \\
  a_{m-M+1}
\end{bmatrix}
\]

Then the solution minimizing \( \epsilon(mT_d) \) is:

\[
W_{M,l} = \left( \sum_{m=1}^{l} A_{M,m}A'_{M,m} \right)^{-1} \left( \sum_{m=1}^{l} A_{M,m}y(mT_d) \right)
\]

For our system, we could obtain channel response readouts through our equipment setup (described in Appendix A). However, we had no signals corresponding to known original data sequences (\( a \) in the above algorithm). Thus, we were forced to recover the original input sequences by ourselves, in essence becoming the data detector.

To recover the data, we used feature length extraction. This technique involves measuring the distance between two consecutive zero crossings and dividing it by the bit interval, \( T \). The sign of the signal between the zero crossings (positive or negative) denotes the original bits (1s or 0s, respectively). Ideally, every pair of zero crossings would
be separated by an exact multiple of $T$. However, in the real channel, factors such as domain bloom (where the pits/lands are written larger or smaller than they should be) and jitter in the write process cause the zero crossing locations to drift from the ideal. Thus, this recovery method is not perfect and requires correction by hand.

One automated technique for improving the automatic recovery accuracy is duty-cycle correction, or DCC. This procedure, described in [26], helps to correct for the baseline drift that causes errors in duty cycle. The algorithm examines the feature lengths over a given interval, compares them to ideal multiples of $T$, and adds or subtracts a constant offset based on the deviations. This procedure greatly improved detection accuracy and the least-squares recovery procedure itself.

With the original data stream extracted, we could perform the least-squares channel identification procedure. The resulting recovered impulse response for a data set of 269 channel bits and channel response oversampled at 7 times the bit rate is shown in Fig. 3.3.

![Figure 3.3: Channel impulse response (least-squares recovery).](image-url)
3.2.2 Channel Impairments

The next important feature of a channel model, once the impulse response is determined, is the collection of impairments that cause the channel output to deviate from its ideal values and thus cause errors in the recovered bitstream. This section will discuss five impairments in the DVD read channel: disc tilt, zero-memory nonlinearity, additive white Gaussian noise, timing jitter, and dropout. Figure 3.4 shows how the impairments enter the system.

![Diagram showing impairments in the channel model.]

Figure 3.4: Impairments in the channel model.

When describing impairments, it useful to define the quantity Signal-to-Noise Ratio (SNR). The SNR of a signal is a measure of the noise power relative to the message signal's power. It is simply defined as the ratio of the mean-square values of the signal and noise, that is:

$$\text{SNR} = \frac{E[S^2]}{E[N^2]}$$

(3.2)

where $S$ is the signal and $N$ is the noise.

The measurement of the performance of a detector in an impaired channel is the Bit Error Rate (BER), defined as:

$$\text{BER} = \frac{N_e}{N_t}$$

(3.3)
where \( N_e \) is the number of errors made by the detector and \( N_t \) is the total number of bits tested.

**Disc Tilt**

Tilt for a DVD system is defined as the angle by which the laser deviates from a perpendicular to the disc readout surface. Tilt can be caused by many factors: errors or motion of the spindle holding the DVD disc in place, improper alignment of the laser read head, and even warping in the DVD disc itself.

Though generally not a factor in CD-ROM system design, disc tilt is a more significant problem in DVD systems. DVD read channels are more sensitive to tilt than CD because of the decreased feature size - the relative effect of tilt is magnified. This has in fact been the impetus for the changeover from slicer based detectors to the more costly but more accurate Viterbi detectors, such as the one described in this thesis.

Any tilt can be decomposed into two fundamental bases: radial tilt, along the direction from the center of the disc to its edge, and tangential tilt, in the instantaneous direction of motion of the disc. Radial tilt increases inter-track interference, while tangential tilt causes increased asymmetry in the impulse response and increased intersymbol interference [26]. Also shown in [26], both forms of tilt increase data-to-clock jitter (tangential tilt more so than radial tilt).

Using the least-squares extracted impulse response means that any tilt in the DVD read channel we are modeling is already present in the impulse response, and does not need to be added to the simulations. Therefore we do not make a comparative study of the performance of the detector in the presence of different amounts of tilt. However, studies [22] have shown that the Viterbi detector improves the detector error rate compared to slicer based detection in the DVD read channel with tilt.
Zero-memory nonlinearity

The concept of a zero-memory nonlinearity (ZNL) as a DVD read channel impairment is introduced in [21]. A ZNL is simply a piecewise linear function that can be used to boost or attenuate the signal amplitude at an arbitrary point in both the positive and negatives senses of the signal. Figure 3.5 shows such an input/output characteristic. The characteristic is applied to the channel data (after convolution with the impulse response) in the simulation to account for magnitude changes in the DVD read channel.

![Zero-memory nonlinearity input/output characteristic.](image)

In the figure, \( \gamma_1 \) and \( \gamma_2 \) are the ordinates for the start of the gain function (\( \gamma_2 < 0 < \gamma_1 \)), while the slopes \( m_1 \) and \( m_2 \) control the amount of gain at each sense (positive or negative).

This piecewise linear transfer function is mathematically described as [21]:

\[
z(x) = \alpha_1 + \alpha_2 \cdot x + \sum_{i=1}^{2} \beta_i \cdot |x - \gamma_i|
\]  

(3.4)
Table 3.1: ZNL simulation parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\gamma_1$</td>
<td>0.597</td>
</tr>
<tr>
<td>$\gamma_2$</td>
<td>-0.656</td>
</tr>
<tr>
<td>$m_1$</td>
<td>1.333</td>
</tr>
<tr>
<td>$m_2$</td>
<td>0.642</td>
</tr>
</tbody>
</table>

where:

\[
\alpha_1 = 0.5 \cdot (\gamma_1 \cdot (1 - m_1) + \gamma_2 \cdot (1 - m_2))
\]

\[
\alpha_2 = 0.5 \cdot (m_1 + m_2)
\]

\[
\beta_1 = -0.5 \cdot (1 - m_1)
\]

\[
\beta_2 = 0.5 \cdot (1 - m_2)
\]

Hwang et. al. ([21]) presents an adaptive version Viterbi algorithm that takes into account the ZNL. For this work, however, we use a time invariant ZNL and precompensate for it in the channel symbol values $s_{i,j}$.

The simulation parameters were determined by using an iterative search to find the values that minimized the mean-squared error (MSE) between the simulation and the sampled DVD. MSE is denoted $\epsilon^2$ and defined as [10]:

\[
\epsilon^2 = \frac{1}{N} \sum_{k=1}^{N} |y_k - \hat{y}_k|^2
\]  

(3.5)

where $y_k$ is the sampled DVD channel value and $\hat{y}_k$ is the corresponding channel simulation value. This yielded the results shown in Table 3.1.

Further evidence of the presence of a ZNL in the DVD read channel data will be presented with the simulation results in section 3.4.

**Additive White Gaussian Noise**

Additive white Gaussian noise, or AWGN, is a common model for noise sources in communication channel simulations. As the name implies, AWGN is an additive form of
noise. The noise values follow a Gaussian probability distribution, which is defined by a mean ($\mu$) and standard deviation ($\sigma$) as shown in Eq. 3.6.

$$n(x) = \frac{1}{\sqrt{2\pi} \cdot \sigma} \cdot e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$ (3.6)

The term white refers to the fact that the noise values are uncorrelated.

The output of a channel only affected by AWGN is:

$$y(t) = x(t) + u(t)$$ (3.7)

where $x(t)$ is the unimpaired channel response and $u(t)$ is uncorrelated noise following a Gaussian distribution.

The AWGN for our experiments was generated with a zero mean and standard deviation calculated to give a desired SNR (e.g. 10 dB).

Timing jitter

Timing jitter is often the parameter used to quantify performance of a DVD read channel. In the literature a jitter of less than 20% relative to the sampling interval is considered sufficient in terms of BER performance for DVD [26].

In our experiments, we generate independent timing offsets using a uniform distribution and apply the results at each sampling interval. Linear interpolation between the points is used to generate the offset signal value. Though linear interpolation does not give the most accurate result, it is fast (for the purposes of simulation) and should in fact yield pessimistic results due to interpolation noise. Thus, it should provide lower bounds for the performance in actual jitter.

Signal dropout

Signal dropout is another common source of error in reading an optical disc. Dropout is commonly caused by flaws introduced to the surface of the disc, such as fingerprints or
scratches. It is characterized by the signal magnitude suddenly attenuating by a large factor (signal energy decreases), so there is no information for the detector to detect. Some implementations, such as [27], have a defect detector that signals a flag when the signal envelope drops below a certain level, signifying dropout. Dropout generally causes burst errors that the RS-PC code is designed to correct.

3.2.3 Final Simulation Model

The structure of the final DVD read channel simulation is shown in Figure 3.6. Though early conceptual designs were done in Matlab, the final simulation was coded in C to improve simulation speed. The C source code is provided in Appendix B.

![Block diagram of DVD read channel simulation model.](image-url)
3.3 Equalization

An important aspect of any communications channel is equalization, attempting to remove as much intersymbol interference and high frequency noise as possible to aid accurate channel detection. Ideally, this is accomplished without further increasing the signal-band noise.

The DVD specification provides a recommended equalizer, consisting of a 6th-order low pass analog Bessel filter (LPF) with a corner frequency (-3 dB point) at 8.2 MHz (to remove high-frequency noise) and a 3-tap transversal filter with coefficients (-0.17, 1.34, -0.17) at times (-2T, 0, 2T) (to boost the high-frequency signal components). All this is designed to operate on the reference channel data rate of 26.16 Mbps. We implemented this exact filter but in digital form. This filter satisfies the specification’s prescribed equalizer characteristics for the 1x channel, which are shown in Table 3.2.

The DVD read channel signals scale linearly with increasing channel speed. For example, signals that are 7 MHz in the 1x channel will be 28 MHz in the 4x channel. Designing the filter in digital form allows the same filter to be used with a data set from any speed channel, so long as the samples are captured at the channel bit rate. The filter is also easily scalable for oversampled data sets.

The simulation filter was designed in Matlab version 5.0, using the Matlab built-in filter functions (indicated here using the typewriter font). The Bessel filter was created using the bessel function, which gives the analog filter coefficients in the s-
domain. This was transformed into a digital filter using the `impinvar` function, and then cascaded with the 3-tap digital filter using the `convolve` function. The resulting digital filter characteristic is shown in Figure 3.7.

![Digital equalizer characteristic](image)

Figure 3.7: Digital equalizer characteristic.

The data we obtained from sampling the DVD player read channel comes after the equalization in that channel implementation. Still, the signal is affected by high frequency noise, which can be removed with the low-pass filter portion of the reference equalizer. All sampled data sets used in this study (e.g. for least squares recovery) are low-pass filtered.

Figure 3.8 shows a comparison of the sampled DVD read channel data before and after low-pass filtering.

Some higher speed channel implementations, such as the 2x channel in [18] and the 4x channel in [27], contain variable gain high-frequency equalizers, which allow boosting of the high frequency components more than the specified 3.2 dB. The detector design in this thesis is based on signals after equalization, so the principles (such as least-squares...
recovery) apply to the output of any equalized channel so long as the output is still linear and time-invariant.

### 3.4 Simulation Results

Using the estimated system impulse response and the recovered user bitstream, we can generate a simulated version of the channel readout and compare the simulation to the original stream. Fig. 3.9 shows this comparison for a short data set.

This process allowed us to verify the accuracy of the ZNL as an impairment in the DVD read channel. As can be seen in Fig. 3.9, the ZNL boosts the high amplitude signals and suppresses the lows.

We can measure the accuracy of our model by examining the error, where error is defined as:

\[
e_k = y_k - \hat{y}_k
\]  

(3.8)
where $y$ is the vector of sampled DVD data and $\hat{y}$ is the simulation vector generated by our model. This error can also be viewed as noise.

Generating histograms for the error signal for both channel models (with and without ZNL) further verifies the ZNL model, as shown in Fig. 3.10.

The Gaussian curves included in Fig. 3.10 are normal curves with mean and standard deviation equal to that of the data set. With the ZNL in place, the noise almost perfectly fits the Gaussian distribution. The noise mean for the ZNL model is -0.0019, standard deviation 0.0537, and the SNR (using the simulation with ZNL as the signal) is 22 dB.

Unfortunately, the noise is not white since a power spectral density (PSD) plot of the noise is not flat across all frequencies, as shown in Fig. 3.11.

This means that AWGN is not a completely accurate model for the remaining impairments in the channel (after the ZNL). The noise up to the channel bit rate (normalized frequency 1) appears to be $1/f$ noise rather than white, while the higher frequency noise (which should be suppressed by the channel equalizer) is white.
CHAPTER 3. SYSTEM DESIGN AND SIMULATION

Figure 3.10: Error histograms for channel simulation, with and without ZNL.

Figure 3.11: Sampled DVD data vs. channel simulation noise power spectral density.
3.5 Summary

In this chapter we discussed:

- The design of a channel model for the DVD read channel. This included the channel impulse response models (Gaussian and least-squares recovery) and the impairments present in the channel for simulating the effects of noise.

- The equalization parameters of the standard DVD read channel, including a digital filter implementation used in this research.

- A comparison of the simulated channel data to actual sampled readout data taken from a DVD player, verifying the accuracy of our channel model.
Chapter 4

Hardware Design of the Detector

4.1 Introduction

This chapter examines the hardware design of the Viterbi detector circuit. We describe the architecture tradeoffs made in implementing the algorithm in hardware, including simulated error performance to justify the choices made. We then discuss the architecture used in the VLSI implementation. Finally, the implementation and specifications of the VLSI chip are presented.

4.2 Design Parameters

In this section, we will examine the design tradeoffs required in making an IC implementation of the Viterbi detector, as opposed to the ideal version of the algorithm. Figure 4.1 shows a block diagram of the detector implementation. We will examine the quantization of analog signals to discrete signals, the branch metric calculation, the number of states in the trellis (i.e. the memory length of the channel), the merge depth required in the survivor memory, and state metric overflow control.
4.2.1 Quantization

Since the Viterbi detector is a digital circuit and the laser output is an analog signal, some kind of analog-to-digital (A/D) conversion is required. The question is: how many bits of precision are required in the A/D converter for accuracy in the detector? This was determined empirically by simulating the detector's BER for various quantization levels, with a timing jitter of 10% and increasing amounts of AWGN (represented by decreasing SNR on the x axis). The number of states is held fixed at 4, which we show in the next section to be a good tradeoff between detector size and BER performance. The outcome of this experiment is shown in Figure 4.2.

Quantization levels of 2 and 3 bits give extremely poor performance, showing a leveling off at a BER of about $7 \times 10^{-3}$. Though 4 bits improves performance compared to 2 or 3 bits (and removes the leveling off), there is still a 1-1.5 dB loss compared to higher quantization levels. At 5 bits, there is a less than 0.5 dB loss compared to higher quantization levels (and a gain of approximately 6 dB compared to the slicer at a BER...
Figure 4.2: BER vs. SNR for varying levels of quantization accuracy ($\nu = 4$, jitter = 10%, merge depth = 25).
of $10^{-3}$, while still being a reasonable size for a hardware implementation. Indeed, [23] claims a flash A/D operating at 4.5x (120 MHz) with 5.5 effective bits of accuracy, which is more than adequate according to our results. Thus we chose this 5 bits for our implementation.

The simulations made two assumptions:

1. The quantizer boundaries are exactly evenly spaced.

2. The quantizer is a saturating quantizer, i.e. it limits any input above the upper input range to the maximum quantized value and any input below the lower input range to the minimum quantized value.

### 4.2.2 Branch Metrics

With the quantization level decided, another tradeoff remains to be made in the branch metric calculation. Though the L2 norm gives the best results in the maximum-likelihood sense (in the presence of AWGN), an alternative with much lower hardware cost is the L1 norm. While the L2 norm is defined as $(y - s_{i,j})^2$, the L1 norm is simply $|y - s_{i,j}|$. When dealing with binary values this results in a great savings in terms of number of bits of storage required and size of arithmetic units.

Given that our incoming signal values are 5 bits wide, the resulting L2 norms can be up to 10 bits wide ($(11111 - 00000)^2 = 1111000001$). Also, L2 norm calculations require a multiplier (to calculate the square of the difference). Multipliers are costly both in terms of hardware area and latency. Finally, state metrics are made up of the sums of past branch metrics. This means the state metric word size must be even larger than that of the branch metrics. Not only the storage, but the adders required for generating new state metrics would also have to be greater than 10 bits wide.

Figure 4.3 shows a simulated run of L1 vs. L2 norm ($v = 4$, quantization = 5 bits, merge depth 25, jitter = 10%). It shows that switching to an L1 norm does not come
without a penalty, but it is small: a loss of less than 0.5 dB compared to the L2 norm over a wide range of SNRs.

![BER vs. SNR for L1 and L2 norm branch metrics](image)

Figure 4.3: BER vs. SNR for L1 and L2 norm branch metrics ($\nu = 4$, quantization = 5 bits, jitter = 10%, merge depth = 25).

### 4.2.3 Number of States

The number of states used in the detector trellis is a determining factor in the size of every block in the Viterbi detector. As discussed earlier, the number of states is $2^\nu$ where $\nu$ is the constraint length.

For a communication channel, the constraint length should be approximately equal to the number of bit intervals where the channel impulse response has a "significant" effect. Significant is unfortunately an imprecise word, and significance varies from channel to channel and from application to application.
Looking at the impulse responses of the DVD read channel, we can see that the magnitude of the response is largest within a $5T$ band around the center of the response in both the Gaussian and extracted impulse responses (as shown in Fig. 4.4).

![Figure 4.4: Impulse responses in terms of channel bit rate.](image)

Knowing this, we hypothesize a constraint length of 4 (since the transitions represent one bit of the overall structure, this takes into account a total of 5 bits) to yield better results than any lower value but not significantly worse results than higher values. Simulations run by varying the number of states and keeping all other factors equal bear out this hypothesis, as shown in Figure 4.5.

Fig. 4.5 shows the BER as a function of the number of states in the Viterbi detector, with slicer detection included as a reference. A constraint length of $\nu = 2$ in fact performs worse than slicer based detection. We hypothesized that this is caused by the zero-memory nonlinearity, since earlier simulations without the ZNL did not have this odd behaviour.

The jump from $\nu = 3$ to $\nu = 4$ is fairly significant (about 2 dB at a BER of $10^{-3}$) but
Figure 4.5: BER vs. SNR for varying numbers of states (no quantization, jitter = 10%, merge depth = 25).
\( \nu = 4 \) is within 0.5 dB of any higher constraint lengths. Thus, as hypothesized, \( \nu = 4 \) gives a good tradeoff between performance and hardware size, as well as improvement of about 6 dB compared to slicer-based detection at a BER of \( 10^{-3} \).

The trellis corresponding to constraint length \( \nu = 4 \), after being reduced according to the \( d = 2 \) run-length limit, is shown in Figure 4.6. The trellis reduces from 16 states \( (2^4) \) to 8, and the number of edges goes down from 32 in the full trellis to 12 in the reduced trellis.

Figure 4.6: Reduced-state trellis for the DVD read channel.
4.2.4 Merge Depth

In the ideal implementation of the Viterbi algorithm, the minimum distance calculation is performed for the entire input sequence. In practice, however, the signal sequences being detected are for all practical purposes infinite in length (DVD discs often contain 4.5-8 Gbytes of data). Thus, we cannot store the survivor path for the entire sequence; after some finite amount of time we must make a (sub-optimal) decision and output a bit.

Fortunately, after a certain amount of time $\delta$ the survivor paths tend to merge into one common ancestor path. All paths are identical after time $\delta$, so we no longer need to store the information but can instead output it. This is the merge depth of the SMU.

The merge depth can affect detector performance, since survivor sequences failing to merge can cause errors in the detected sequence. A general rule for merge depth is approximately 4-6 times the constraint length of the design for moderate SNR. Since our chosen constraint length is $\nu = 4$, this implies a merge depth of 16-24 deep. To ensure merging, we chose to slightly exceed the upper value and use a merge depth of 25.

To validate the merge depth, simulations were run which monitored the merge point in the survivor memory at a worst-case SNR of 5 dB. The system parameters decided upon to this point (5 bit quantization, $\nu = 4$, L1 norm) were used in the simulation, as well as timing jitter of 5%. The results after over 20 000 data points show merging to occur after no more than 15 trellis stages, less than our chosen merge depth.

4.2.5 Survivor Memory Management

There exist several algorithms for storing and updating the surviving sequences as determined by the trellis decisions. The earliest and conceptually simplest method is the register exchange algorithm, while a later alternative is one of the various forms of the traceback algorithm (originally introduced as the pointer method in [28], and summarized
with newer techniques in [29]).

The register exchange algorithm is a simple physical mapping of the channel’s trellis into a hardware circuit, with add-compare-select (ACS) units at each trellis node. For each state in the trellis there is one storage register, and decisions from the state metric calculations dictate from which of its two inputs the register stores its new value. The number of stages in a register exchange implementation corresponds to the merge depth. For example, Fig. 4.7 shows a register exchange unit for a 2 state trellis.

![Diagram of register exchange implementation for 2 state trellis.]

Figure 4.7: Register exchange implementation for 2 state trellis.

Traceback algorithms, on the other hand, do not store the actual bit values output by the trellis. Instead, they store pointers indicating which branch of the trellis was taken. Traceback algorithms require 3 stages of operation:

- Decision write, which writes these pointers into the traceback memory.
- Traceback read, which decodes these pointers back to the merge depth.
- Decode read, which decodes and outputs the bits past the merge depth.

The three basic traceback algorithms are \( k \)-pointer even, \( k \)-pointer odd, and one-pointer. Details on them can be found in [29].
There are tradeoffs in choosing the best algorithm for survivor management. Register exchange is conceptually simpler and easier to implement. It also has lower latency than the three traceback algorithms in [29] for a given merge depth. The traceback algorithms, on the other hand, consume significantly less power than the register exchange algorithm. Register exchange requires one read and one write for every register in every clock stage, resulting in a total bandwidth $T \times 2^r$ (where $T$ is the depth of the unit). Traceback algorithms only require a write bandwidth of $2^r$ and read bandwidth of $k$ (where $k$ is the number of pointers). Also, traceback algorithms use standard SRAM for their pointer storage. In the fabrication process used for this chip, the SRAM is only rated to a speed of 80 MHz, less than the targeted 104.64 MHz for this chip. We would have had to design custom low-latency SRAM cells, a project outside the scope of this thesis.

For its simpler implementation, lower latency, and feasibility in our given fabrication process, we decided to use the register exchange algorithm in our architecture.

### 4.2.6 State Metric Overflow

State metric overflow is another challenge faced in the hardware implementation of the Viterbi algorithm. In the ideal case, the state metrics grow unbounded as required. However, in the real implementation, a finite number of bits are available for storing the state metrics, so some form of normalization to restrict their dynamic range is required.

For our implementation, we use most significant bit clearing as described in [20]. In this technique, the state metric storage is made 3 bits wider than the input bit size (thus 8 in our case). The most significant bits of each state metric are monitored with an AND tree structure, and when all MSBs are equal to one they are cleared back to zero. This has the effect of subtracting $2^r$ from each state metric. In this way the state metrics are continually normalized to prevent overflow.

There is a non-zero probability that one state metric could overflow before the MSBs of all the other states are set. In practice we expect the likelihood of this to be extremely
low—it never occurred in our simulations of 1 million points. Further, if this case does occur the results will not be catastrophic, and should cause a negligible number of detector errors.

4.2.7 Final Results

The final simulated performance of the detector, using all the parameters discussed previously, is shown in Fig. 4.8

![Graph showing BER vs SNR](image)

Figure 4.8: BER vs SNR for final simulation ($\nu = 4$, $q=5$ bits, merge depth = 25, L1 norm, state metric overflow).

At a bit error rate of $10^{-3}$, the Viterbi detector implementation has about a 6 dB advantage over the traditional slicer based method.
CHAPTER 4. HARDWARE DESIGN OF THE DETECTOR

4.3 Chip Architecture

We will now discuss the implementation of the VLSI Viterbi detector for the DVD read channel. This section will present block diagrams describing the designs used in the architecture as shown back in Fig. 4.1.

4.3.1 Branch Metric Generator

The branch metric generator (BMG) is responsible for generating the branch metrics, the edge weights for the trellis. As decided by our experiments, the BMG was designed using an L1 norm and 5-bit wide input value. The channel symbol values (s values) were set to the values used for the simulations; however, the circuit was created with the ability to shift in new s values through the input lines. This allows us to adjust the branch metrics, for example if improved values are determined or the circuit is used in a channel with a different impulse response.

The BMG is made up of an array of blocks, each of which calculates the branch metric for one trellis edge. Fig. 4.9 shows one of these component blocks of the BMG.

With the symmetrical Gaussian impulse response used in our original simulations, several edges in fact have the same branch metric calculations (the same s values), so the redundant ones could be eliminated. However, as stated above we designed the circuit to be able to shift in new s values if needed. Since these might come from an asymmetric impulse response, the decision was made to leave the extra comparison units in. This is
especially important when the least-squares recovery method is used, since the extracted impulse response is asymmetrical.

The branch metric blocks are connected together to form the BMG, as shown in Fig. 4.10.

As stated previously, the $s$ values can be left as the default reset values or shifted in through the input port if new values are desired. Fig. 4.11 shows the circuit for this function. In this implementation, the reset values correspond to 5-bit quantized versions of the Gaussian impulse response, and are enumerated in Table 4.1.

### 4.3.2 Add-Compare-Select Unit

The add-compare-select, or ACS, unit is the computational core of the Viterbi algorithm. Each ACS subnode corresponds to one of the states in the trellis diagram, accepting two branch metric values, adding them to the state metric, and storing the smaller result as the new state metric.

The choice of surviving branch (0 or 1, in this implementation) is signaled by the decision lines, labeled $\eta_i(k)$. These values are used in the survivor memory unit, discussed
Figure 4.11: Comparison value storage.

Table 4.1: Default branch metric reference values.

<table>
<thead>
<tr>
<th>State/Branch</th>
<th>Reference Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0/0</td>
<td>00001</td>
</tr>
<tr>
<td>0/1</td>
<td>00100</td>
</tr>
<tr>
<td>1/1</td>
<td>01011</td>
</tr>
<tr>
<td>3/1</td>
<td>10100</td>
</tr>
<tr>
<td>7/0</td>
<td>11000</td>
</tr>
<tr>
<td>7/1</td>
<td>11011</td>
</tr>
<tr>
<td>8/0</td>
<td>00100</td>
</tr>
<tr>
<td>8/1</td>
<td>00111</td>
</tr>
<tr>
<td>12/0</td>
<td>01011</td>
</tr>
<tr>
<td>14/0</td>
<td>10100</td>
</tr>
<tr>
<td>15/0</td>
<td>11011</td>
</tr>
<tr>
<td>15/1</td>
<td>11110</td>
</tr>
</tbody>
</table>
in section 4.3.3.

The ACS unit also incorporates the state metric storage. The result is a feedback loop, with the previous time slice's state metrics feeding back through the storage to the current time slice's input. Since there is feedback, the stage cannot be pipelined; thus it generally represents the critical path in a Viterbi algorithm implementation. Although algorithms exist for improving speed through such techniques as block processing (see [30]), it was determined that this would not be required to achieve our speed goals of 4x.

As with the BMG, the ACS can be broken down into computational blocks that are combined to form the whole unit. Figure 4.12 shows one of these ACS subblocks.

These blocks are combined, one for each state in the trellis, to form the entire unit as shown in Fig. 4.13.

The normalization blocks for preventing state metric overflow use the design shown in Fig. 4.14.

4.3.3 Survivor Memory Unit

Finally, the survivor memory unit (SMU) is responsible for storing the survivor sequences decided upon by the ACS. We used the register exchange algorithm for the SMU in this implementation.
Figure 4.13: Add-compare-select unit.
As with the other units examined so far, the SMU can be broken up into component pieces which can be connected to form the whole unit. In this case, one slice of the SMU (corresponding to one stage in the trellis) is shown in Fig. 4.15.

These are simply combined in \( n \) stages, where \( n \) is the merge depth (25 in our case). Figure 4.16 shows the completed unit. The initial values correspond to the bit being shifted out of the state. The final output (detected channel value) is taken to be the output of state 0 at the final time slice. Not shown here is the fact that this reduces the hardware somewhat, since storage not used in the path leading to state 0 at the final time slice can be removed.

### 4.3.4 Clock Doubling

The IMS LogicMaster XL-60 IC tester available at the University of Toronto has a maximum clock speed rating of 60 MHz [31]. Therefore, to test the chip at 104.64 Mbps (4x speed) some clock doubling circuitry is required. We generate a doubled clock using a simple XOR circuit with two 90° out-of-phase input clocks. As well, since the inputs must come in at the clock rate, a swinging buffer is used to switch between two 5-bit wide input lines. The resulting circuitry is shown in Fig. 4.17.

The input and output buffers swing on the rising edge of the doubled clock, meaning that each input/output is held for one entire \( \phi_1 \) clock cycle.
Figure 4.15: Survivor memory unit slice.

Figure 4.16: The survivor memory unit.
Since the clock doubling circuit is an XOR, the chip can operate with a single clock by holding $\phi_2$ low and using $\phi_1$ as the only clock.

Figure 4.18 shows the timing diagram for the main inputs and outputs to the chip: the two clock phases, $\phi_1$ and $\phi_2$, the two inputs to the swinging input buffers, $IN_1$ and $IN_2$, and the two data outputs $OUT_1$ and $OUT_2$.

The clock doubling circuitry should permit the chip to be tested in the IMS tester at
Table 4.2: IC parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process</td>
<td>TSMC 0.35 μm CMOS, 3 metal, dual poly</td>
</tr>
<tr>
<td>Core area</td>
<td>0.62 mm²</td>
</tr>
<tr>
<td>Total area</td>
<td>5.413 mm²</td>
</tr>
<tr>
<td>Supply voltage</td>
<td>5 V</td>
</tr>
<tr>
<td>Total pins</td>
<td>32</td>
</tr>
<tr>
<td>Packaging</td>
<td>68 pin PGA</td>
</tr>
<tr>
<td>Target speed</td>
<td>104.64 Mbps</td>
</tr>
<tr>
<td>Power consumption</td>
<td>approx. 100 mW</td>
</tr>
</tbody>
</table>

a maximum rate of 120 MHz, which is greater than our targeted speed.

4.4 Chip Implementation Specifications

The architecture presented in the previous section was implemented using the VHDL hardware description language. Full VHDL source code can be found in Appendix C.

The resulting description was translated into hardware resources using the Synopsys design system, and implemented in 0.35 μm CMOS technology using the Cadence IC design system. The Synopsys synthesized schematic was verified by comparing it to the C simulation output using 1 million randomly generated test vectors. The chip is currently being fabricated by the Canadian Microelectronics Corporation (CMC) at TSMC. A plot of the chip layout is shown in Fig. 4.19.

The parameters of the resulting chip are summarized in Table 4.2.

The final pinouts of the packaged chip won’t be known until it’s returned by CMC; instead, a list of pins and their functions is given in Table 4.3.
Figure 4.19: Mask layout for Viterbi detector.
Table 4.3: IC pin list and functions.

<table>
<thead>
<tr>
<th>Pin</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>input[0...4]</td>
<td>Input lines (one half of swinging input buffer).</td>
</tr>
<tr>
<td>input_b[0...4]</td>
<td>Input lines (other half of swinging input buffer).</td>
</tr>
<tr>
<td>scan_bm</td>
<td>Channel symbol value scan-in control. While held low, channel symbol values are scanned in through the inputs (input and input_b) and stored in the reference memory.</td>
</tr>
<tr>
<td>reset</td>
<td>Resets the chip to an all zeros state and loads default channel symbol values when held low for one clock cycle.</td>
</tr>
<tr>
<td>test_se</td>
<td>Enables serial scan-in of multiplexed flip-flop test vectors generated automatically in Synopsys.</td>
</tr>
<tr>
<td>test_si</td>
<td>Input port for serial scan-in in test mode.</td>
</tr>
<tr>
<td>clk_phi1</td>
<td>Primary clock port.</td>
</tr>
<tr>
<td>clk_phi2</td>
<td>Secondary clock port. Hold at zero for single clock operation, switch at same rate as clk_phi1 but 90 degrees out-of-phase for clock doubled operation.</td>
</tr>
<tr>
<td>output[0...4]</td>
<td>Output lines (one half of swinging output buffer). Also doubles as the test scan output port.</td>
</tr>
<tr>
<td>output_b[0...4]</td>
<td>Output lines (other half of swinging output buffer).</td>
</tr>
</tbody>
</table>
4.4.1 Test

The chip includes test_se and test_si, scan enable and scan in pins. These interface with built in multiplexed flip-flop automatic test circuitry (inserted by Synopsys) which allows us to test for faults using the test vectors generated by the automatic test pattern generation (ATPG) section of the Synopsys software. The test coverage achieved with the ATPG vectors is greater than 95%. The scan results are received from port output. These combined with the randomly generated test vectors used in the functional testing of the circuit will allow us to verify the chip’s operation in the IMS tester.

4.4.2 Critical Path

Using an ideal clock (no uncertainty, rise and fall times of 0) the critical path of the detector is, as expected, in the ACS loop from the output of the BMG to a state metric storage register. Simulations with the ideal clock show the detector able to run at a maximum speed of greater than 6x (where 6x is 156.96 MHz). Adding timing delays and clock skew reduces the maximum speed, but further simulations including these impairments indicate the chip should operate at 4x speed (104.64 MHz).

4.5 Summary

In this chapter we:

- Discussed the architecture design decisions made in implementing the Viterbi detector in hardware, including constraint length, quantization, merge depth, survivor memory management, branch metric calculation, and overflow control.

- Described an architecture for a VLSI implementation of the architecture using the decisions made in the previous section.
• Presented an implementation of the above architecture, including a layout plot for a 0.35 μm CMOS implementation and a description of the operating parameters of the chip.
Chapter 5

Conclusions

5.1 Summary and Conclusions

The threshold method of data detection works well for DVD drives at the reference (1x) data rate. However, as channel bit rates increase, impairments cause the threshold detector to produce an unacceptable number of errors. Improved detection methods, such as those employing the Viterbi algorithm, are required.

This thesis has presented the system design of a Viterbi detector for the DVD read channel and a VLSI implementation of that design capable of running at 104.64 Mbps (4x) speeds.

Chapter 2 provided some of the background material and theory used in the thesis. A brief introduction to the DVD format was introduced including the DVD read channel. We then gave an overview of data detection methods suitable for the DVD read channel. Next, a discussion of reduced state trellises for run-length limited coded channels was presented. We finished the chapter with a discussion of existing literature on Viterbi detectors for the DVD read channel.

In chapter 3 we discussed a simulation model of the DVD read channel, including a channel model, equalization methods, and a comparison of our simulation to data
sampled from a DVD player.

Chapter 4 introduced the hardware architecture for the detector. We presented a summary of the design decisions made, then described the resulting architecture. This chapter concluded with a description of the specifications of the resulting VLSI chip being manufactured in a 0.35 μm CMOS process.

5.2 Contributions

The contributions of this thesis are as follows:

- A read channel model for the DVD system, including impulse response and channel impairments.
- A system architecture for a Viterbi detector suitable for the DVD read channel.
- A VLSI implementation of the above architecture.

5.3 Suggestions for Future Research

This thesis is simply the first step in a full DVD read channel implementation. There are many areas that still need addressing before the design could be used in an actual DVD player circuit. We present some of the issues here.

5.3.1 Analog Interfacing

The detector presented in this thesis was designed purely in the digital domain. The real detector will have to operate in an analog channel receiving analog signals from the laser read head. We showed that the detector performs with an acceptable error rate using a quantization of 5 bits. However, this means that for the 4x channel we require an A/D converter capable of operating at 104.64 Mbps. Also, circuitry such as a phase-locked
loop (PLL) is required for timing recovery to ensure the sampling of the laser readout data occurs at the correct instant. A full analog interface, including A/D converter and timing recovery, would be an interesting area for further study. The analog 4x read channel IC described in [19] and [27] with an A/D converter could make an excellent front-end for this research.

### 5.3.2 Equalization

This study simply used the reference DVD equalizer in its design. However, this equalizer was designed for threshold detection at 1x speeds. Further research could examine equalizers, either analog or digital, to better match the characteristics of the channel response to maximum likelihood detection.

### 5.3.3 Demodulation and Decoding

Detection is simply the first step in the DVD read channel. Further research could examine the implementation of an EFMPlus demodulator and ECC decoder chip (such as those in [12] or [11]) to combine with this work. This would complete the DVD read channel chain and output the original user bits.

### 5.3.4 Higher Data Rates

In this thesis we implemented a detector for the 4x (104.64Mbps) data rate. However, published designs already exist running at 4.5x [23], and retail products exist operating at 6x [3] [4]. Further research using high speed Viterbi algorithm implementations (such as those described in [30]) on the detector structure presented in this thesis or more advanced fabrication processes (such as 0.25 μm) could push the detection rate even higher.
5.3.5 Soft Detection

The Viterbi detector presented here is a hard output detector. Soft output detectors include not only the detected bit value, but also a measure of the reliability of that bit. Perhaps a soft decision detector, such as the soft-output Viterbi algorithm (SOVA) [32] [33] or the forward-backward algorithm [34], could possibly be used in EFMPlus demodulation (such as described in [35]) or be passed through the EFMPlus demodulator to the RS-PC decoder to improve bit error rate performance.
Bibliography


Appendix A

Measurement Setup

Equipment:

- Akai DVP-1000 DVD player.

- Textronix TDS 220 100 MHz digital oscilloscope (1 Gs/s).

- Hewlett-Packard 1141A differential probe using HP 1142A probe control and power module.

The equipment was configured as shown in Fig. A.1

The data taken from pins 48 & 49 of the AN862FBQ IC is the data just before slicing occurs, as shown in the block diagram of Fig. A.2. The connection from the input bias to the slicer is internal to the chip, so we had to take our samples at the point shown.

The data from the Textronix TDS 220 was passed over a GPIB bus interface to a measurement workstation, which recorded the measured values into ASCII text files.

The TDS 220 scope is capable of capturing a maximum of 2500 data points. Table A.1 shows the sampling rates and amount of data captured, both in ns and bits (where bits is calculated as \( \text{time} \times \frac{1}{T} \), and \( T \) is the channel bit interval given that the channel bit rate for the DVP-1000 is 27 Mbps; \( T = 37.04 \text{ ns} \).)
Figure A.1: Measurement equipment setup.

<table>
<thead>
<tr>
<th>Sampling Period</th>
<th>Amount of Data Captured</th>
<th>Bits Captured</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 ns</td>
<td>10 ms</td>
<td>270</td>
</tr>
<tr>
<td>10 ns</td>
<td>25 ms</td>
<td>675</td>
</tr>
<tr>
<td>20 ns</td>
<td>50 ms</td>
<td>1350</td>
</tr>
</tbody>
</table>
Figure A.2: Block diagram of IC read channel, including measurement point.

The three different ranges allowed us a good tradeoff between number of bits captured and temporal resolution.
Appendix B

C Source Code for Channel Model
/* add_jitter */
* Add clock jitter to a signal by linear interpolation */
* AUTHOR
* Kenneth Chalmers
* Department of Electrical and Computer Engineering
* University of Toronto
* COPYRIGHT
* Copyright 1999 Kenneth Chalmers
*/

#include <math.h>
#include <stdlib.h>
#include "add_jitter.h"
#include "random.h"

int add_jitter(double* array, int numpts, double jitter_pct)
{
    double jitter;
    double unitcol;
    double delta_T;
    double new_point;
    double* new_array;
    int i;

    // Jitter fraction in %100 */
    jitter = jitter_pct / 100.0;

    // Array to hold new points */
    new_array = (double*)malloc(sizeof(double) * numpts);
    for(i = 0; i < numpts; i++)
    {
        // Calculate amount of jitter to add to this sample */
        unitcol = (double)random() / (double)RANDOM_RANGE;
        delta_T = ((unitcol * 2) - 1) * jitter;
        // If defined(DESIGN)
        printf("Delta T = %.4f\n", delta_T/100);
        endif
        if(delta_T == 0)
            new_point = array[i];
        else
            // Linearly interpolate to find response at jitter point */
            // Calculate interpolation parameters */
            double x1, x2, y1, y2, m, b;
            x1 = i;
            y1 = array[i];
            if(delta_T > 0)
            {
                x2 = i+1;
                if(i+1 >= numpts)
                    y2 = 0;
                else
                    y2 = array[i+1];
            }
            else
            {
                x2 = i-1;
                if(i-1 < 0)
                    y2 = 0;
                else
                    y2 = array[i-1];
            }
            // Calculate slope and intercept for interpolation */
            m = (y2 - y1) / (x2 - x1);
            b = y2 - m*x2;
            // Perform interpolation */
            new_point = m * (delta_T + x1) + b;
        }
        new_array[i] = new_point;
    }
    memcpy(array, new_array, sizeof(double) * numpts);
    free(new_array);
    return 0;
}

int binrv (unsigned int seed, /* initialisation seed */
           int reset) /* set to 1 to restart */
{
    int rv;
    long int random ();
    extern char *initstate (), *setstate ();
    static char *state[BYTES], *pivsetate();
    static int flag;
    if (reset == 1) {
        flag = 10;
        return (0);
    }
    /* binrv */
    return (0);
if (flag == LO)
{  
    prevstate = initstate (seed, state, BYTES);  
    flag = HI;  
    return (0);  
}  
else  
{  
    prevstate = setstate (state);  
    rv = random (flag & 01);  
    return (rv);  
}

/******************************************************************************
/* $Include <stdlib.h> $Include "bitget.h" 
int *  
bitget (long num, int as)
/*  
* Produces the reversed bit stream corresponding to n, in an array.
* The values in the array are bipolar (-1 and 1)
* num is the number to be converted.
* as is the number of bits to return.
*/
{
    int i, *a, b;
    long n;
    n = num;
    a = (int *) malloc (as * sizeof (int));
    for (i = 0; i < as; i++)  
    {  
        if (n & 1)  
            b = 1;
        else  
            b = -1;
        a[i] = b;
        n = n >> 1;
    }
    return a;
}

/******************************************************************************
/* $Include <stdlib.h> $Include "bitget.h" 
int *  
bitget (long num, int as)
/*  
* Produces the reversed bit stream corresponding to n, in an array.
* The values in the array are bipolar (-1 and 1)
* num is the number to be converted.
* as is the number of bits to return.
*/
{
    int i, *a, b;
    long n;
    n = num;
    a = (int *) malloc (as * sizeof (int));
    for (i = 0; i < as; i++)  
    {  
        if (n & 1)  
            b = 1;
        else  
            b = -1;
        a[i] = b;
        n = n >> 1;
    }
    return a;
}

/******************************************************************************
/* $Include <stdlib.h> $Include "bitget.h" 
int *  
bitget (long num, int as)
/*  
* Produces the reversed bit stream corresponding to n, in an array.
* The values in the array are bipolar (-1 and 1)
* num is the number to be converted.
* as is the number of bits to return.
*/
{
    int i, *a, b;
    long n;
    n = num;
    a = (int *) malloc (as * sizeof (int));
    for (i = 0; i < as; i++)  
    {  
        if (n & 1)  
            b = 1;
        else  
            b = -1;
        a[i] = b;
        n = n >> 1;
    }
    return a;
}

/******************************************************************************
/* $Include <stdlib.h> $Include "bitget.h" 
int *  
bitget (long num, int as)
/*  
* Produces the reversed bit stream corresponding to n, in an array.
* The values in the array are bipolar (-1 and 1)
* num is the number to be converted.
* as is the number of bits to return.
*/
{
    int i, *a, b;
    long n;
    n = num;
    a = (int *) malloc (as * sizeof (int));
    for (i = 0; i < as; i++)  
    {  
        if (n & 1)  
            b = 1;
        else  
            b = -1;
        a[i] = b;
        n = n >> 1;
    }
    return a;
}
Appendix B. C Source Code for Channel Model

```c
void channel_encode(int input, int* output)
{
    int length = input;
    int output_length = 0;
    int byte = 0;
    int bit = 0;
    int output_bit = 0;
    int i;

    for (i = 0; i < length; i++)
    {
        byte = (input[i] & 0x0F) << 4;
        output_bit = (output[i] & 0x0F) << 4;
        for (bit = 0; bit < 5; bit++)
        {
            if (byte & (1 << bit))
                output_bit |= (1 << (bit + 4));
        }
        output[i] = output_bit;
        output_length += 1;
    }
    return output_length;
}
```
{  
  FILE* out;
  size_t written;

  /* Open file for writing */
  out = (FILE*)fopen(filename, "wb");
  if(out == NULL)
    return FIO_OPENERR;

  /* Store length of array (simplifies reading later) */
  written = fwrite((char*)&len, sizeof(int), 1, out);
  if(written != 1)
    {  
      fclose(out);
      return FIO_WRITEERR;
    }

  /* Store array */
  written = fwrite(array, sizeof(double), len, out);
  if(written != len)
    {  
      fclose(out);
      return FIO_WRITEERR;
    }

  fclose(out);
  return FIO_OK;
}

/*
 * Read an array of doubles
 */
int read_double_array(char* filename, double** array, int* len)
{  
  FILE* inf;
  size_t read;

  /* Open file for reading */
  inf = (FILE*)fopen(filename, "rb");
  if(inf == NULL)
    return FIO_OPENERR;

  /* Read length of array */
  read = fread((char*)&len, sizeof(int), 1, inf);
  if(read != 1)
    {  
      fclose(inf);
      return FIO_READERR;
    }

  /* Allocate memory for array */
  *array = (double*)malloc(sizeof(double) * *len);
  if(*array == NULL)
    {  
      fclose(inf);
      return FIO_ALLOCERR;
    }

  /* Read array */
  read = fread(*array, sizeof(double), *len, inf);
  if(read != *len)
    {  
      fclose(inf);
      return FIO_READERR;
    }

  return FIO_OK;
}

/*
 * Read an array of integers from an ASCII file
 */
#define ALLOCSIZE 1000
int read_int_array_asc(char* filename, int** array, int* len)
{  
  int num_items;
  FILE* inf;
  int* read_ptr;

  /* Open file for reading */
  inf = (FILE*)fopen(filename, "rt");
  if(inf == NULL)
    return FIO_OPENERR;

  /* Allocate initial memory bit */
  *array = (int*)malloc(sizeof(int) * ALLOCSIZE);
  if(*array == NULL)
    {  
      fclose(inf);
      return FIO_ALLOCERR;
    }

  /* Read away */
  num_items = 0;
  read_ptr = *array;
  while(!feof(inf))
    /* Read array */
    read = fread((char*)read_ptr, sizeof(int), ALLOCSIZE, inf);
    if(read != ALLOCSIZE)
      {  
        fclose(inf);
        return FIO_READERR;
      }

  *len = num_items;

  return FIO_OK;
}
APPENDIX B. C SOURCE CODE FOR CHANNEL MODEL

```c
{ /* Read a data point */
    read_point = &array[num_items];
    fscanf(in, "%d", &read_point);
    if(ferror(in))
    {
        free(array);
        fclose(in);
        return FIO_READERR;
    }
    num_items++;

    /* Check if we've filled up the buffer, reallocate if true */
    if(num_items >= ALLOCSIZE)
    {
        array = (int*)realloc(array, sizeof(int)*ALLOCSIZE*2*(num_items/ALLOCSIZE + 1));
        if(array == NULL)
        {
            free(array);
            fclose(in);
            return FIO_ALLOCPED;
        }
    }
    fclose(in);
    *len = num_items;

    /* Realocate pointer to exact correct size */
    array = (int*)realloc(array, sizeof(int)*num_items);
    if(array == NULL)
    {
        free(array);
        return FIO_ALLOCPED;
    }
    return FIO_OK;

    /* Error strings */
    char* fio_err = strerror(errno);
    switch(fio_err)
    { case FIO_OK:
      return "Everything ok";
      case FIO_OPENERR:
      return "Could't open file";
      case FIO_READERR:
      return "Problem during read";
      case FIO_WRITEERR:
      return "Problem during write";
      case FIO_ALLOCPED:
      return "Memory allocation error";
      default:
      return "Unknown file I/O error";
    }
}

/******************************************************************************
************ random number generator - gaussian pdf *****************
#include <math.h>
#include "rand.h"

#define RANGE 1147483647.0  /* (2^31)-1 */
#define BYTES 256
#define HI 1
#define LO 0

double

// allocate list z, m, unsigned int seed, int reset
{
    int count;
    long long random();
    return char* initstate(unsigned char* initstate, char* int)
    float random();
    return static int flag;
    static char state[BYTES];
    *prevstate;
    if (reset == HI)
    {
        flag = LO;
        return (0.0);
    }
    if (flag == LO)
    {
        prevstate = initstate(seed, state, BYTES);
        flag = HI;
        return (0.0);
    }
    else
    {
        prevstate = initstate(state);
        sum = 0.0;
        for (count = 1; count <= 4; count++)
        {
            unifl = random() / RANGE;
            sum = sum + unifl;
        }
        rv = ((sum - 24.0) / 2.0) * z + m;
        return (rv);
    }
```

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <math.h>
#include "spot-jitter.h"
#include "snr.h"
#define ZNL_ON

int main(int ac, char** av)
{
    char *spotfn;
    char *bfn;
    double *bm_a;
    double *spot_a;
    int nu;
    int spottag, i;
    int overs;
    if(ac != 4)
    {
        printf("Usage: gen_bm <spot_file> <nu> <b_file>\n\n");
        return 5;
    }
    spotfn = av[1];
    nu = stoi(av[2]) + 1;
    bfn = av[3];
    i = read_spot(spotfn, &spot_a, &spottag, &overs);
    if(i < 0)
    {
        printf("Exiting...");
        exit(5);
    }
    spot_start = [spot_a[spottag/2]];
    for(i = 0; i < nu; i++)
    {
        printf("%g\n", spot_start[i]);
    }
    /* Calculate branch metric values from spot profile */
    bm_a = bvals(nu, spot_start);
    if defined(ZNL_ON)
    /* Use zero-memory nonlinearity to add distortion */
    fprintf(stderr, "Using ZNL\n");
    for(i = 0; i < (i < nu); i++)
    bm_a[i] = xnl(bm_a[i]);
    endif
    /* Write branch metric values to file */
    WriteBMVal(bfn, bm_a, nu);
    free(bm_a);
    free(spot_a);
    return 0;
}

int main(int ac, char** av)
{
    int i;
    int nums = 1 << (conlength-1);
    bfn = (FILE*)fopen(bfn, "w");
    if(bfn == NULL)
    {
        printf("Error writing to branch metric file \n", bfn);
        return -1;
    }
    else
    {
        fprintf(bfn, "%d\n", conlength-1);
        for(i = 0; i < nums; i++)
        fprintf(bfn, "%d %g\n", bm_a[2*i], bm_a[2*i]+1);
    }
    fclose(bfn);
    return 0;
}
APPENDIX B.

Source Code for Channel Model

```c
// APPENDIX B. C SOURCE CODE FOR CHANNEL MODEL

#include <stdio.h>
#include <stdlib.h>
#include "randombits.h"
#include "ntri.h"
#include "conv.h"
#include "lai.h"
#include "esplus_encode.h"
#include "fileio.h"
#include "util.h"

#define SNR_ON "Zero-memory nonlinearity (noise source)"

#define cochpaper 1111111111111111111111111111111111

int main(int ac, char** av) {
    char* spotfn;
    int numbits; int numodbits;
    char* refoutfn;
    char* readdoutfn;
    int* input_a;
    int* modulated_a;
    char* ntri_a;
    double* ntripb_a;
    double* output_a;
    int i;

    for(i = 5) {
        printf("Usage: gen_input <spot file> # of data bits <reference output file> read out file>\n"); return 5;
    }

    // Parse parameters
    spotfn = av[1];
    numbits = stoi(av[2]);
    refoutfn = av[3];
    readdoutfn = av[4];

    // Make sure 0 of bits is a multiple of 8, for EFM encoding?
    numbits /= 2;

    // Since the number of bits doubles after EFM
    numodbits = 2*numbits;
    numodbits *= 2;

    // 16 output bits for every 0 in input

    // Channel simulation

    // Generate random input bits
    input_a = randbits(numbits, numodbits);

    // Dump the pre-modulation bits
    fprintf(stderr, "Dumping pramod.txt\n");
    write_int_array_asc(pramod.txt, input_a, numbits);

    // EFM modulate the bits
    modulated_a = esplus_modulate(input_a, numbits);

    // Dump the bits to a file
    fprintf(stderr, "Dumping rdm.txt\n");
    write_int_array_asc(rdm.txt, modulated_a, numodbits);

    // NRII encode the bits
    ntri_a = ntri(numodbits, modulated_a);

    // Write the NRII encoded bits to the reference file
    fprintf(stderr, "Writing reference file to\n");
    write_int_array_asc(refoutfn, ntri_a, numodbits);

    // Convert NRII array to doubles
    nrti_a = norm_double_array(ntri_a, numodbits);

    // Convolve NRII waveform with spot response
    fprintf(stderr, "Doing ISI modeling with linear shift register\n");
    output_a = linear_shift ISI_file(nrti_a, numodbits, spotfn);

    if defined(SNR_ON)
        // Use zero-memory nonlinearity to add distortion
        fprintf(stderr, "Using SNR\n");
        for(i = 0; i < numodbits; ++i)
            output_a[i] = sli(output_a[i]);

    // Write final channel simulation to data output file
    fprintf(stderr, "Writing final output to\n");
    write_double_array(readdoutfn, output_a, numodbits);

    // Deallocate all memory
    free(output_a);
    free(input_a);
    free(refoutfn);
    free(ntri_a);
    free(nrti_a);
    free(ntripb_a);
    free(output_a);
}
```

93
```c
f1nrrldoubls,a)
j
trorldulate~~~~
fl
input-a)
return
1
*
UrltrHRZIArrryi
Urite
M
rrrry
O!
lntogrrr
tllr
a/
lnt
Mlt~ZIArriy(chir*
Ln,
Int*rrry, Int
len)
FIU*
rtl-fi
Int
ir
toril
8
Oi
1
<
lmi
i**l
(lt(rrrry(1l
r
O)
tprlntt
(nrti-t,
*]
*)i
elee
Iprlnttinrtf,!
'0
9
l
I
tcloirlnrrl-t)
return
01
1
/
IntloLbublrArrry:
Convrrt
an
irrry
O!
lnttgrr
vrluri to rn
irrry
of doubler
doublrrrryi
nta
irrry, int
lm)
1
doubla*
doublrrrry
lnt
II
doublrrrry
= [double*)ulloc
[lrsot
(doublr)
lan)]
If[doublerrrry
4i
NVLL)
I
prlntt
lWError
illocrtlnp
mraory
for
doublr
rrriy
0
) return
HVLL~
1
raturn
doublerrrryr
1

/******************************************************/
/
/*
/* ISI.C:
*/
/* Stands for "Linear shift-register ISI", models the ISI of the channel
/* using the impulse response and a linear shift register. Should be
/* 'orders of magnitude' faster than the old convolution method.
/*
/* AUTHOR
/* Kenneth Chalmers
/* Department of Electrical and Computer Engineering
/* University of Toronto
/*
/* COPYRIGHT
/* Copyright 1999 Kenneth Chalmers
/* */
/*include <malloc.h>
#include "isi.h"
#include "spot_jitter.h"
#include "gauss.h"
#include "random.h"
#include "read_spot.h"

#if defined(UNIX)
#include "stdio.h"
#endif

#define SEED 1

double
linear_shift_isi(double* waveform, int wavebits, double t0, double datacrate, double isibit,
t, double amplitude, double jitter_pct, int oversample)
{

double* shiftrreq;
double* output;
int outbits;
double* hbit;
int hbits;
int i, j;
double T, delta_T, uintol;

/* For Gaussian spot profile. */
hbits = (3*isibits) + 1;

/* Calculate # of output bits (taking into account shift register size) */
outbits = wavebits + (1 + hbits);

/* Allocate the memory */
output = (double*)malloc(sizeof(double) * outbits);

/* Create shift register to hold input data */
shiftrreq = (double*)malloc(sizeof(double) * hbits);

/* Initialize shift register to zeros (all -1) state */
for(i = 0; i < hbits; i++)
shiftrreq[i] = -1;

/* Sampling period: data rate in Mb/s, sampling rate in ns */
T = 1.0 / datacrate * 1e3;

/* Truncate waveform: */
for(i = 0; i < wavebits; i++)
waveform[i] = waveform[i] * amplitude;

/* Write impulse response to ISI file */
for(i = 0; i < len; i++)
if(array[i] > 0)
fprintf(nrrl_f, "1
\n");
else
fprintf(nrrl_f, "0
\n");

fclose(nrrl_f);
return 0;
}

/* IntToDoubleArray:
*/
/* Convert an array of integer values to an array of doubles
*/
double* IntToDoubleArray(int* array, int len)
{

double* doublesarray;
int i;

doublesarray = (double*)malloc(sizeof(double) * len);
if(doublesarray == NULL)
{
printf("Error allocating memory for double array\n");
return NULL;
}

for(i = 0; i < len; i++)
doublesarray[i] = (double)array[i];
return doublesarray;
}

/* WriteNRRArray:
/* Write an array of integers to a file
*/
int WriteNRRArray(char* fn, int* array, int len)
{
FILE* nrrl_f;
int i;
nrrl_f = (FILE*)foopen(fn, "wt");
if(nrrl_f == NULL)
{
printf("Error opening NRR file \n\n");
return -1;
}

for(i = 0; i < len; i++)
if(array[i] > 0)
fprintf(nrrl_f, "1\n");
else
fprintf(nrrl_f, "0\n");

fclose(nrrl_f);
return 0;
}
```
/* Initialize Gaussian random number generator */
gauze(0.0, 0.0, SEED, 1);

/* Generate the output stream */
for(i = 0; i < output_bits; i++)
{
    double sum = 0.0;
    /* Calculate amount of jitter to add to clock */
    /* OLD: Gaussian: delta_T = gauze(jitter_pct * T1/3.0, 0.0, SEED, 0); */
    unitorl = (double)random() / (double)(RAND_MAX);  
    delta_T = (unitorl * 2.0 - 1.0) * T * jitter_pct;
}

if defined(DBG)
    print("T = %lg, Delta T = %lg (%d)%n", T, delta_T, (delta_T/T)*100);
#endif

/* Generate Gaussian spot profile */
    h = spot(h, gausrate, lstbits, delta_T, oversample);
for(j = 0; j < hbitlength; j++)
    h[j] = amplitude;

for(j = 0; j < hbitlength; j++)
    sum += h[j] * shiftreg[j];

output[i] = sum;

/* Shift forward contents of shift register */
for(j = hbitlength-1; j > 0; j--)
    shiftreg[j] = shiftreg[j-1];

/* Shift in new bit (pad with zeros after waveform) */
if(i < wavetable)
    shiftreg[0] = waveform[i];
else
    shiftreg[0] = -1;

free(h);

return output;

/*
 * linear_shift_isi_file
 *
 * Like linear_shift_isi, but reads spot profile from
 * a file instead of generating it.
 */

double*
linear_shift_isi_file(double* waveform, int wavetable_bits, char* spotfn)
{
    double* shiftreg;
    double* output;
    int output_bits;
    double* h;
    int hbitlength;
    int hshiftbits; /* Size of shift register */
    int hovers;    /* No, not flying -- oversampling factor in h */
    int i, j;

    /* Read spot profile */

    /* Initialize shift register to zeros (all -1) state */
    for(i = 0; i < hshiftbits; i++)
        shiftreg[i] = -1;

    /* Allocate the memory */
    output = (double*)malloc(sizeof(double) * output_bits);

    /* Create shift register to hold input data */
    shiftreg = (double*)malloc(sizeof(double) * hshiftbits);

    /* Initialize shift register to zeros (all -1) state */
    for(i = 0; i < hshiftbits; i++)
        shiftreg[i] = -1;

    /* Generate the output stream */
    for(i = 0; i < output_bits; i++)
    {
        double sum = 0.0;

        /* Generate output using shift register */
        for(j = 0; j < hshiftbits; j++)
            sum += h[j] * shiftreg[j];

        if(i > (hshiftbits/2))
            output[(i-(hshiftbits/2))-1] = sum;

        /* Shift forward contents of shift register */
        for(j = hshiftbits-1; j > 0; j--)
            shiftreg[j] = shiftreg[j-1];

        /* Shift in new bit (pad with zeros after waveform) */
        if(i < wavetable)
            shiftreg[0] = waveform[i];
        else
            shiftreg[0] = -1;

        free(h);

        return output;
    }

/*
 * noisify
 *
 * Add an MQ signal to make it's SNR match a target value
 *
 * AUTHOR
 * Kenneth Balmer
 * Department of Electrical and Computer Engineering
 * University of Toronto
 *
 * COPYRIGHT
 */
# include <stdio.h>
# include <malloc.h>
# include <math.h>
# include "noisy.h"
# include "gauss.h"

#define SEED 0x0D /* Seed for Gaussian number generator */

/* noisy: * After an input block to have a given SNR */
void noisy(double* block, int len, double SNR)
{
    double noise; /* Array of noise values */
    double sigpow; /* Signal power of input block */
    int i;

    /* Allocate memory for noise block */
    noise = (double*)malloc(sizeof(double) * len);
    if (noise == NULL)
        { printf("Not enough memory for noise block\n");
            return;
        }

    /* Initialise random number generator */
    gauss(0, 0, SEED, 1);

    /* Calculate the signal power of the block */
    sigpow = signal_power(block, len);
    /* printf(stderr, \"Sig: \%f\n\", sigpow); */

    /* Generate a noise block for the given signal power and SNR */
    gen_noise_block(noise, len, sigpow, SNR, 1);

    /* Apply the noise to the input block */
    for (i = 0; i < len; i++)
        block[i] += noise[i];

    /* De-allocate memory */
    free(noise);
}

double gen_noise_block(double* noise, int noise_len, double sigpow, double snr, double code_rate)
{
    int j;
    double noisePower, stddev;

    /* Generate a block of noise */
    for (j = 0; j < noise_len; j++)
        { noise[j] = gauss(1.0, 0.0, SEED, 0);
            noisePower = signal_power(noise, noise_len);
        }

    /* Normalise the noise to unit variance */
    for (i = 0; i < noise_len; i++)
        { noise[i] /= sqrt(noisePower);
            stddev = sqrt(sigpow / code_rate * pow(10.0, snr / 10.0));
            noise[i] *= stddev;
            return stddev;
        }

    double signal_power(double* block, int len)
    {
        int i;
        double pwr = 0;
        for (i = 0; i < len; i++)
            pwr += pow(block[i], 2.0);
        return pwr / len;
    }

    /***************************************************************************/
    /***************************************************************************/
    /***************************************************************************/

    #include <malloc.h>
    #include "nrzi.h"

    int *
    nrzi (int sz, int *inBits)

    /***************************************************************************/
    /***************************************************************************/
    /***************************************************************************/

    int *
    nrzi (int sz, int *inBits)

    /***************************************************************************/
    /***************************************************************************/
    /***************************************************************************/

    /***************************************************************************/
    /***************************************************************************/
    /***************************************************************************/

    /***************************************************************************/
    /***************************************************************************/
    /***************************************************************************/
for (i = 0; i < nu; i++)
{
    if (imbits[i] == 1)
        state = state;
    if (state == 1)
        a[i] = 1;
    else
        a[i] = -1;
}
return a;
}

/****************************************************************************
** prune_trellis:
** Routines to prune the trellis for the DVD Viterbi decoder to
** conform to a particular d constraint. For standard DVD players,
** d = 2. Note that the information in the trellis is NRZI encoded,
** so simply checking for n consecutive zeros is not a correct test.
** Note the use of -1 as an 'invalid branch' flag. This can be used
** since we'll be quantizing all values to around 5-6 bits in general,
** and -1 would only be a problem at 32 bits, so we can use it here.
**
** AUTHOR
** Kenneth Chalmers
** Department of Electrical and Computer Engineering
** University of Toronto
**
** COPYRIGHT
** Copyright 1999 Kenneth Chalmers
**
** */

#include "quantization.h"
#include "viterbi_12.h"

int violates_d(int d, int nu)
{
    int i,
    int num_zeros = 0,
    int num_ones = 0;
    int last_bit, cur_bit, i;
    last_bit = input;
    for(i = 0; i < nu; i++)
    {
        cur_bit = (cur_state & (1 << i)) == 0 ? 0 : 1;
        if(cur_bit != last_bit) /* if i then this bit is a 1 (NRZI encoding) */
        {
            if(num_zeros < d)
            {
                printf("Violated: %d %d\n", num_zeros, num_ones); /*
                return -1; /* Runlength limit violation!!! */
            }
            else /* We have a valid isolated 1 */
            {
                num_zeros = 0;
                num_ones = 0;
                if(num_ones > 1 && d >= 0) /* Special case: if d = 0 */
                {
                    printf("Violated: %d %d\n", num_zeros, num_ones);
                    return -1; /* Runlength limit violation!!! */
                }
            }
        }
        else /* cur_bit == last_bit, therefore this bit is a zero */
        {
            num_ones += 1;
        }
        last_bit = cur_bit;
    }
return 0;
}
void quant_reset(int bits, double low, double high)
{
    int i;
    double rangesize; /* The distance between range boundaries (see code) */

    /* Set variables - make sure inlow <= inhigh */
    qbits = bits;
    inlow = min(low, high);
    inhigh = max(low, high);

    /* Calculate number of range values for given qbits */
    /* Note the -1; if there are 4 ranges, only 3 values are needed */
    /* (the top and bottom values aren't used since the inputs are saturating) */
    qranges = (int)(pow(2, qbits) - 1);

    /* DEBUG */
    printf("%d ranges\n", qranges);

    /* Free old range memory, allocate new */
    free(grangevals);
    grangevals = (double*)malloc(sizeof(double) * qranges);

    /* Calculate the size of the ranges */
    rangesize = (high - low)/(double)(qranges+1);

    /* Set up ranges */
    for(i = 0; i < qranges; i++)
    {
        grangevals[i] = low + (i)*(rangesize);
    }

    /* DEBUG: Print range */
    /*
    * if(i > 0)
    *   print("Range %d to %g to %g\n", i, grangevals[i-1], grangevals[i]);
    */
}

/*
* quantize
*
* Quantise an input given the parameters passed to quant_reset
*/

int quantize(double input)
{
    int out = 0; /* Final output value */
    int i;

    /* Cycle through ranges, stop when input fits in a range */
    for(i = 0; i < qranges; i++)
    {
        if(input < grangevals[i])
            break;
        out++;
    }

    /* Internal state variables */
    static int qbits; /* Number of quantization bits */
    static double inlow; /* Low value in the input dynamic range */
    static double inhigh; /* High value in the input dynamic range */
    static double* grangevals; /* Values which demark the ranges */
    static int qranges; /* Number of ranges in grangevals */

    /* Quantize */
    /* Reset the quantiser to operate for a given # of bits and range */
    */

    /* Useful macros */
    #define min(x, y) ((x) ? (x) : (y))
    #define max(x, y) ((x) > (y) ? (x) : (y))

    /* SOURCE CODE FOR CHANNEL MODEL*/
    /* */
    /* Quantize a floating point value to an integer range */
    /* of a certain number of bits. Given the input range and the number */
    /* of bits to quantize to, these routines will output the quantized */
    /* values corresponding to the given inputs. For example; */
    /* */
    /* Input range of -1 to 1 */
    /* Quantise to 2 bits (i.e. 4 ranges) */
    /* -1 <= x < -0.5 --> y = 00 */
    /* -0.5 <= x < 0 --> y = 01 */
    /* 0 <= x < 0.5 --> y = 10 */
    /* 0.5 <= x <= 1 --> y = 11 */
    /* */
    /* Values below the low value of the range are clipped to the lowest */
    /* output, and values above the highest value of the range are clipped */
    /* to the highest output. Thus, in the example above, -2 would be clipped */
    /* to 0, while 1.5 would be clipped to 0. */
    /* */
    /* Effectively this simulates a saturating A/D converter */
    /* */
    /* AUTHOR*/
    /* Kenneth Chalmers */
    /* Department of Electrical and Computer Engineering */
    /* University of Toronto */
    /* */
    /* COPYRIGHT */
    /* Copyright 1999 Kenneth Chalmers */
    /* */
    #include <math.h>
    #include <stdio.h>
    #include "quantize.h"
    #include "stdio.h"
} /* Return the calculated output */
return out;

}/**
/ * Copyright (c) 1985 Regents of the University of California.
/ * All rights reserved. The Berkeley Software License Agreement
/ * specifies the terms and conditions for redistribution.
/ */

#if defined(LIBC_SCCS) & defined(_lint)
static char *std[] = {"#include", "5.2 (Berkeley) 3/9/86"};
#endif /* LIBC_SCCS and not lint */
#include <stdio.h>
#include "random.h"

/* random.c *
* An improved random number generation package. In addition to the standard
* rand()/srand() like interface, this package also has a special state info
* interface. The generate() routine is called with a seed, an array of
* bytes, and a count of how many bytes are being passed in; this array is then
* initialized to contain information for random number generation with that
* much state information. Good sizes for the amount of state information are
* 32, 64, 128, and 256 bytes. The state can be switched by calling the
* setstate() routine with the same array as was initialized with initstate().
* By default, the package runs with 128 bytes of state information and
* generates far better random numbers than a linear congruential generator.
* If the amount of state information is less than 32 bytes, a simple linear
* congruential R.N.G. is used.
* Internally, the state information is treated as an array of longs; the
* second element of the array is the type of R.N.G. being used (small
* integer); the remainder of the array is the state information for the
* R.N.G. Thus, 32 bytes of state information will give 7 longs worth of
* state information, which will allow a degree seven polynomial. (Note: the
* second word of state information also has some other information stored
* in it -- see setstate() for details.)
* The random number generation technique is a linear feedback shift register
* approach, employing trinomials (since there are fewer terms to sum up that
* way). In this approach, the least significant bit of all the numbers in
* the state table will act as a linear feedback shift register, and will have
* period 2^deg - 1 (where deg is the degree of the polynomial being used,
* assuming that the polynomial is irreducible and primitive). The higher
* order bits will have longer periods, since their values are also influenced
* by pseudo-random carries out of the lower bits. The total period of the
* generator is approximately deg*(2^deg - 1); thus doubling the amount of
* state information has a vast influence on the period of the generator.
* Note: the deg*(2^deg - 1) is an approximation only good for large deg,
* when the period of the shift register is the dominant factor. With deg
* equal to seven, the period is actually much longer than the 7*(2^7 - 1)
* predicted by this formula.
*/

/* For each of the currently supported random number generators, we have a
* break value on the amount of state information (you need at least this
* many bytes of state info to support this random number generator), a degree
* for the polynomial (actually a trinomial) that the R.N.G. is based on, and
* the separation between the two lower order coefficients of the trinomial. */

define TYPE_0 0 /* linear congruential */
define BREAK_0 8
#define DBG_0 0
#define SEP_0 0
#define TYPE_1 1 /* x**7 * x**3 + 1 */
define BREAK_1 33
#define DBG_1 7
#define SEP_1 3
#define TYPE_2 2 /* x**15 + x + 1 */
define BREAK_2 64
#define DBG_2 15
#define SEP_2 1
#define TYPE_3 3 /* x**31 * x**1 */
define BREAK_3 128
#define DBG_3 31
#define SEP_3 3
#define TYPE_4 5 /* x**63 + x + 1 */
define BREAK_4 256
#define DBG_4 63
#define SEP_4 1

/* Array versions of the above information to make code run faster -- relies
* on fact that TYPE_i == i. */
define MAX_TYPES 5 /* max number of types above */
static int degrees[MAX_TYPES] =
(DBG_0, DBG_1, DBG_2, DBG_3, DBG_4);
static int steps[MAX_TYPES] =
(SEP_0, SEP_1, SEP_2, SEP_3, SEP_4);

/* Initially, everything is set up as if from:
* initstate(1, srandth, 128);
* Note that this initialization takes advantage of the fact that rand() advances the front and rear pointers 10^*rand_deg times, and hence the
* rear pointer which starts at 0 will also end up at zero; thus the soroeth
* element of the state information, which contains info about the current
* position of the rear pointer is just
* MAX_TYPES2*(ptr - state) + type_i == type_i. */
static long randdbl[64L * 1] =
    (TYPE3,
     0x4a319073, 0x532d8c04, 0x6b4c1182, 0x55e1f342,
     0x6a9b31e0, 0x6d66e8e5, 0x3d30c2d0, 0x48f3461b,
     0x7e4b9b68, 0x5eb0d2b0, 0x3b5c951b, 0x3b6554c0,
     0x4e2e48f6, 0x8eb8979f, 0x4b1e0c70, 0x3d4b3b6b,
     0x4d71e6a, 0x35826a86, 0x6a07557d, 0x5a0f3557,
     0xd7f1b0d6, 0x5fa56051, 0x6b1e6069, 0x8a94e6fc,
     0x3e411f3f, 0xc0223c86, 0x5e42ab6, 0x8e4d77b,
     0x6c5a9d76, 0x0999222b, 0xc924b9b);

    /*
    * fptr and rptr are two pointers into the state info, a front and a rear
    * pointer. These two pointers are always rand_wptr places apart, as they cycle
    * cyclically through the state information. (Yes, this does mean we could get
    * away with just one pointer, but the code for random() is more efficient this
    * way). The pointers are left positioned as they would be from the call
    * initstate(1, randbl, 128)
    * (The position of the rear pointer, rptr, is really 0 (as explained above)
    * in the initialization of randbl) because the state table pointer is set
    * to point to randbl[1] (as explained below).
    */

static long *fptr = randbl[64L * 1];
static long *rptr = randbl[64L * 1];

    /*
    * The following things are the pointers to the state information table,
    * the type of the current generator, the degree of the current polynomial
    * being used, and the separation between the two pointers.
    * Note that for efficiency of random(), we remember the first location of
    * the state information, not the search. Hence it is valid to access
    * state[-1], which is used to store the type of the R.N.G.
    * Also, we remember the last location, since this is more efficient than
    * indexing every time to find the address of the last element to see if
    * the front and rear pointers have wrapped.
    */

static long *state = randbl[1];
static int rand_type = TYPE3;
static int rand_deg = DBL3;
static int rand_wptr = SEP3;
static long *rand_ptr = randbl[DBL3 * 1];

    /*
    * random:
    * Initialize the random number generator based on the given seed. If the
    * type is the trivial no-state-information type, just remember the seed.
    * Otherwise, initialize state[] based on the given 'seed' via a linear
    * congruential generator. Then, the pointers are set to known locations
    * that are exactly rand_wptr places apart. Lastly, it cycles the state
    * information a given number of times to get rid of any initial dependencies
    * introduced by the L.C.R.N.G.
    * Note that the initialization of randbl[] for default usage relies on
    * values produced by this routine.
    */

void random(unsigned X) {

    register int i, j;
    if (rand_type == TYPE0) {
        state[0] = x;
    }
    else {
        j = 1;
        state[0] = x;
        for (i = 1; i <= rand_deg; i++) {
            state[i] = 1103515245 * state[i - 1] + 12345;
        }
        fptr = state[rand_wptr];
        rptr = state[0];
        for (i = 0; i < 10 * rand_deg; i++)
            rand_ptr[i];
        random();
    }

    /*
    * initstate:
    * Initialize the state information in the given array of n bytes for
    * future random number generation. Based on the number of bytes we
    * are given, and the break values for the different R.N.G.'s, we choose
    * the best (largest) one we can and set things up for it. random() is
    * then called to initialize the state information.
    * Note that on return from initstate(), we set state[-1] to be the type
    * multiplied with the current value of the rear pointer; this is so
    * each time we call initstate(0) won't lose this information and will
    * be able to restart with the same initial state.
    * Note: the first thing we do is save the current state, if any, just like
    * initstate() so that it doesn't matter when initstate is called.
    * Returns a pointer to the old state.
    */

    char* initstate(unsigned seed, /* seed for R. N. G. */
                    char* arg_state, /* pointer to state array */
                    int n) /* # bytes of state info */
    {
        register char*state = (char*) (*state[-1]);
        char* arg_state,
        /* pointer to state array */
        int n) /* # bytes of state info */
    {
        register char*state = (char*) (*state[-1]);
        char* arg_state, /* pointer to state array */
        int n) /* # bytes of state info */
    {
        register char*state = (char*) (*state[-1]);
        char* arg_state, /* pointer to state array */
        int n) /* # bytes of state info */
    {
        register char*state = (char*) (*state[-1]);
        char* arg_state, /* pointer to state array */
        int n) /* # bytes of state info */
    {
        register char*state = (char*) (*state[-1]);
        char* arg_state, /* pointer to state array */
        int n) /* # bytes of state info */
[ rand_type = TYPE_1;
    rand_deg = 32;1);
    rand_sep = SEP_1;
} else
{ if (n < BREAK_J)
{ rand_type = TYPE_3;
    rand_deg = 64;
    rand_sep = SEP 3;
} else
{ if (n < BREAK_4)
{ rand_type = TYPE_4;
    rand_deg = 128;
    rand_sep = SEP_4;
} else
{ rand_type = TYPE_5;
    rand_deg = 256;
    rand_sep = SEP_5;
} }
}]
state = *((long *) argv_state)[0]; /* first location */
end_ptr = state[rand_deg]; /* must set end_ptr before random */
random (end_ptr);
if (rand_type == TYPE_0) state[1] = rand_type;
else state[1] = MAX_TYPES * (rand_type - state) + rand_type;
return (state);

/* setstate:
* Restore the state from the given state array.
* Note: it is important that we also remember the locations of the pointers
* in the current state information, and restore the locations of the pointers
* from the old state information. This is done by multiplexing the pointer
* location into the same word of the state information.
* Note that due to the order in which things are done, it is OK to call
* setstate() with the same state as the current state.
* Returns a pointer to the old state information.
*/
char* setstate(char* argv_state)
{ register long *new_state = (long *) argv_state;
  register int type = new_state[0] % MAX_TYPES;
  register int rear = new_state[0] / MAX_TYPES;
  char *o state = (char*) (state[1]);
  if (rand_type == TYPE_0) state[1] = rand_type;
  else
  { state[1] = MAX_TYPES * (rand_type - state) + rand_type;
    switch (type)
    { case TYPE_0: case TYPE_1: case TYPE_2: case TYPE_3:
      rand_type = type;
      rand_deg = degree(type);
      rand_sep = sep[type];
      break;
    default:
      fprintf(stderr, "setstate; state info has been munged; not changed.\n"");
      state = new_state[1];
      if (rand_type != TYPE_0)
      { wrptr = state[rear];
        wrptr = state[rear + rand_sep % rand_deg];
      end_ptr = state[rand_deg]; /* set end_ptr too */
      return (o state);
      }
    }
  state = *((long *) argv_state)[1]; /* first location */
  end_ptr = state[rand_deg]; /* must set end_ptr before random */
  random (end_ptr);
  if (rand_type == TYPE_0) state[1] = rand_type;
  else state[1] = MAX_TYPES * (rand_type - state) + rand_type;
  return (state);

/* random:
* If we are using the trivial TYPE_0 R.N.G., just do the old linear
* congruential bit. Otherwise, we do our fancy trinomial stuff, which is the
* same in all that other cases due to all the global variables that have been
* set up. The basic operation is to add the number at the rear pointer into
* the one at the front pointer. Then both pointers are advanced to the next
* location cyclically in the table. The value returned is the sum generated,
* reduced to 31 bits by throwing away the 'least random' low bit.
* Note: the code takes advantage of the fact that both the front and
* rear pointers can't wrap on the same call by not testing the rear
* pointer if the front one has wrapped.
* Returns a 31-bit random number.
*/
long random()
{ long i;
  if (rand_type == TYPE_0) { i = state[0] + state[0] + 1103515245 + 12345) & 0x7fffffff;
  } else
  { *okrptr += *rptr;
    i = (*okrptr + 1) & 0x7fffffff; /* chucking least random bit */
    if (++o PTR + end_ptr)
      { fptr = state;
        ++rptr;
      } else
      { if (++rptr == end_ptr)
        rptr = state;
      }
    }
  state[1] = MAX_TYPES * (rand_type - state) + rand_type;
  switch (type)
  { case TYPE_0: case TYPE_1: case TYPE_2: case TYPE_3:
    rand_type = type;
    rand_deg = degree(type);
    rand_sep = sep[type];
    break;
  default:
    fprintf(stderr, "setstate; state info has been munged; not changed.\n"");
    state = new_state[1];
    if (rand_type != TYPE_0)
    { wrptr = state[rear];
      wrptr = state[rear + rand_sep % rand_deg];
    end_ptr = state[rand_deg]; /* set end_ptr too */
    return (state);
  }
  state = *((long *) argv_state)[1]; /* first location */
  end_ptr = state[rand_deg]; /* must set end_ptr before random */
  random (end_ptr);
  if (rand_type == TYPE_0) state[1] = rand_type;
  else state[1] = MAX_TYPES * (rand_type - state) + rand_type;
  return (state);
  }
APPENDIX B. C SOURCE CODE FOR CHANNEL MODEL

```c
#include <malloc.h>
#include "randombits.h"
#include "binrv.h"

int *
randombits (int seedl, int num)

/* Produces a random bitstream.
   seedl is the random number generator seed value to use
   num is the number of bits to produce.
   Returns an array of random bits
*/
{
    int i, *a;
    a = (int *) malloc (num * sizeof (int));
    i = binrv (seedl, 1);  /* reset binrv */
    for (i = 0; i < num; i++)
        a[i] = binrv (seedl, 0);
    return a;
}

#include <malloc.h>
#include "randombits.h"
#include "binrv.h"

int *
rdouble (int seedl, int d, int num)

/* Produces a random bitstream, corresponding to a d+2 constraint
   seedl is the random number generator seed value to use
   d is the minimum number of zeros between one's
   num is the number of bits to produce.
   Returns an array of random bits */
{
    int i, j, *a;
    a = (int *) malloc (num * sizeof (int));
    for (i = 0; i < d; i++)
        a[i] = 0;
    i = binrv (seedl, 1);  /* reset binrv */
    for (i = d; i < num - d; i++)
        a[i] = binrv (seedl, 0);
    if (a[i] == 1)  /* d constraint dictates that any '1' is followed */
        { /* by at least d '0's */
            for (j = i + 1; j <= i + d; j++)
                a[j] = 0;
            i = i + d;  /* set next d inputs to 0 */
        }
    for (i = num - d; i < num; i++)  /* set trailing d bits to 0 in case last generated */
        a[i] = 0;  /* bit was 1 */
    return a;
}
```

```c
#include <malloc.h>
#include "randombits.h"
#include "binrv.h"

int *
read_spw

/* Read the spot profile from a file for use in branch metric/input
   file generation. Spot file should be ASCII in the form: *
   * Number of points in spot (integer) *
   * Oversampling factor (integer) *
   * Point 1 (double) *
   * ... *
   * Point n (double) *
   * <SPW> *
   * INPUTS *
   * spotin - Filename of spot file *
   * spot_a - Pointer to unallocated double array to receive spot profile *
   * spoplen - Pointer to integer, receives length of spot profile *
   * oversam - Pointer to integer, receives oversampling factor *
   * RETURNS *
   * 0 on success, nonzero on failure */
```
#include <stdio.h>
#include <stdlib.h>
#include "read_spot.h"

int read_spot(char *spotfn, double *spot, int *spotlen, int *oversam) {
  FILE *spot_file = fopen(spotfn, "r");
  if(spot_file == NULL) {
    return -1;
  }

  /* Read the spot length and oversampling factor */
  fscanf(spot_file, "##dimensions", spotlen, oversam);

  /* Allocate memory for the spot array */
  *spot_a = (double*)malloc(*spotlen * sizeof(double));
  if(*spot_a == NULL) {
    fclose(spot_file);
    return -2;
  }

  /* Read the spot array */
  for(i = 0; i < *spotlen; i++) {
    fscanf(spot_file, "##spot", &(*spot_a)[i]);
    if(feof(spot_file)) {
      printf(stderr, "premature end of spot file\n");
      fclose(spot_file);
      return -3;
    }
  }

  fclose(spot_file);
  return 0;
}

double * spot (double t0, double datarate, int isibits, double delta, int oversample) {
  double T, Tm, Tt, *result, multfactor, expfactor;
  int numsamples, int centre, int i, int j;

  numsamples = ((2 * isibits + 1) * oversample); T = 1 / datarate * 1e3; /* datarate is in Mbits/sec, T is in ns */
  Tm = T / (double)oversample; /* Oversampling factor reduces single period */
  t = (double *)malloc (numsamples * sizeof(double));
  result = (double *)malloc (numsamples * sizeof(double));
  centre = (oversample * isibits);
  for (i = 0; i < numsamples; i++) {
    t[i] = ((i - centre) * T) + delta; /* delta shifts the whole response */
    multfactor = 2 * T / (T0 * sqrt(Pi)); /* T factor inferred */
    for (i = 0; i < numsamples; i++) {
      expfactor = (2 * t[i] / T0) * (2 * t[i] / T0);
      result[i] = multfactor * exp(-1 * expfactor);
    }
  }

  return result;
}

/* gaussian spot intensity profile */
#include <stdio.h>
#include <math.h>
#include <malloc.h>
Complete simulation of a Viterbi detector, with optional quantization of the inputs and all internal data paths.

For quantization, compile with QUANTIZATION_ON defined (i.e.,
gcc -DQUANTIZATION_ON ...)

* AUTHOR
  * Kenneth Chalmers
  * Department of Electrical and Computer Engineering
  * University of Toronto

* COPYRIGHT
  * Copyright 1999 Kenneth Chalmers
  *
*/

#include <limits.h>
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

if defined(QUANTIZATION_ON)
  * Warning Quantization is on
  * include "viterbi_q.h"
else
  * Warning Quantization is off
  * include "viterbi.h"
#endif

#define.dgv(x) ((x) * (x))

//define MERGE_DEPTH_TEST
//define INTERNAL_SIZE_TEST

/* Use abs(diff) (L2 norm) instead of diff^2 in metric calc. */
#define RN ABS DIFF

static BIT** survivors;  /* Survivor sequence memory */
static int survivor_len;  /* Length of survivor sequence memory */
static BIT* decisions;    /* Buffer to store decision bits */
static int mru;           /* Length of code memory */
static int num_states;    /* Number of states in algorithm [2^m] */
static STATE* state[2];  /* Double-buffer for state recording */
static STATE* cur_state; /* Pointer to current state */
static int next_state;    /* Index of next state */
static BRANCH_METRIC* branch_metrics; /* Branch metric values for each state */

/* Internal function prototypes */
int acc(INPUT g);
int is_valid_branch(int state, int branch);
void normalise_state(void);
void advance_survivors(void);
INTPUT branch_metrics(int state, int branch, INTPUT g);
int defined(MERGE_DEPTH_TEST)
int merge_depth(void);
#endif

/* Initialize Viterbi detector with given parameters */

int survivor_len = survivor_len_in;
for(i = 0; i < states; i++)
  survivors[i] = (BIT*)alloc(sizesof(BRANCH_METRIC));
if(survivors == NULL)
  fprintf(stderr, "Error allocating survivor memory\n");
exit(5);

/* Survivor memory */
for(i = 0; i < states; i++)
  survivors[i] = (BIT*)alloc(sizesof(BIT*));
if(survivors == NULL)
  fprintf(stderr, "Error allocating survivor memory\n");
exit(5);

/* Decision bit memory */
for(i = 0; i < states; i++)
  decisions[i] = (BIT*)alloc(sizesof(BIT*));
if(decisions == NULL)
  fprintf(stderr, "Error allocating decision memory\n");
exit(5);

/* Current/new state storage */
for(i = 0; i < 2; i++)
  state[i] = (STATE*)alloc(sizesof(STATE));
if(state[i] == NULL)
  fprintf(stderr, "Error allocating state memory \n");
exit(5);

/* Hard-coded initial survivor state values */
/* Clear the rest of the survivor memory */
for(j = 1; j < survivor_len; j++)
  survivors[i][j] = 0;
if(l == (states/2))
  j = 1;
else
  j = 0;
/*
 * Copy branch metrics
*/
memcpy(branch_metrics, branch_metrics_in, sizeof(BRANCH_METRIC) * states);
/* Misc. parameters */
um_states = states; /* Number of trellis states */
nu = nu_len; /* Constraint length */
cur_state = state[0]; /* Current state storage (which buffer) */
next_state = 1; /* Next state buffer */
return l;
}

/* Free all memory allocated in start_viterbi */

void finish_viterbi()
{
  int i;
  /* Memory deallocation */
  /* Branch metric memory */
  free(branch_metrics);
  /* Survivor memory */
  for(i = 0; i < num_states; i++)
    free(survivors[i]);
  survivors = 0;
  /* Decision bit memory */
  free(decisions);
  decisions = 0;
  /* Current/new state storage */
  for(i = 0; i < 2; i++)
    { free(state[i]);
      state[i] = 0;
    }
}

/* Perform actual Viterbi detection operation for one input/output */
# !branch_valid
if (branch_valid)
  #if defined(QUANTIZATION_ON)
    printf("%4d (Q5d): %5d %5d / %5d %5d ",
      #if defined(QUANTIZATION_WON)
        lbranchvalid,
      #else
        print("%4d (Q5p): %5.4f %5.4f / %5.4f %5.4f ",
      #endif
      1, 0,
      branch_metric(1=1, 1, 2, 0), branch_metric(1=1, 1, 2, 0),
      cur_state(1=1, 1, 2, 0), cur_state(1=1, 1, 2, 0));
  #endif
#endif

/* Select operation: determines which path survives */
if (p1 < p2)
  /* Record state selection for this state */
  state[next_state][1] = p1;
  decisions[1] = 0;
  if (p1 < p_min)
    p_min = p1;
  shortest = 1;
}
else
  state[next_state][1] = p3;
  decisions[1] = 1;
  if (p2 < p_min)
    p_min = p3;
  shortest = 1;
}

#elif defined(DEBUG)
  printf("\n");
#endif

t++;
/* Switch to new state */
cur_state = state[next_state];
next_state = 1;
#elif defined(DEBUG)
  printf("State Metrics:\n");
  for (i = 0; i < num_states; i++)
    if (is_valid_branch(i, 0) || is_valid_branch(i, 1))
      #if defined(QUANTIZATION_ON)
        printf("%4d: %4d \n", i, cur_state[i], i == shortest ? 'x' : ' ');
      #else
        printf("%4d: %4d \n", i, cur_state[i], i == shortest ? 'x' : ' ');
      #endif
      #elif (getchar() == 'q') exit(10);
#endif

/* print("%4d\n", shortest); */

return shortest;

/** Check for invalid branch flag. Returns 0 on invalid branch, else non-0 */
int is_valid_branch(int state, int branch)
{
  /* Use -999 flag to indicate an invalid branch - NOT SAFE in general, but works for this application */
  return branch_metrics[state].metric_value(branch) * -999;
}

/* Branch metric generator. */

INPUT branch_metric(int state, int branch, INPUT g)
{
  #if defined(INVALID_BRANCH)
    return (INPUT)(fabs(g - branch_metrics[state].metric_value(branch)));
  #else
    /* Normal branch metric calculation */
    return (INPUT)(sqrt(g - branch_metrics[state].metric_value(branch)));
  #endif
}

/* State normalization, to prevent overflow */
void normalize_states()
{
  STATE p_min;
  #if defined(INTERNAL_SIZE_TEST)
    STATE p_max;
  #endif
  int 1;

  #if defined(QUANTIZATION_ON) /* And MSB style normalization */
  extern int QBITS;
  p_min = 1 << (QBITS+2);
  for(i = 0; i < num_states; i++)
    { /* Check that MSB is set - if not on ANY input, then don't normalize */
      if((cur_state[i] & (1 << (QBITS+3))) == 0)
        p_min = 0;
      if((cur_state[i] & (1 << (QBITS+3))) == 1)
        fprintf(stderr, "Warning! State metric overflow!");
    }
  #else
    /* Find the minimum state value for normalization */
    p_min = cur_state[0];
    for(i = 1; i < num_states; i++)
      if(cur_state[i] < p_min)
        p_min = cur_state[i];
  #if defined(INTERNAL_SIZE_TEST)
  */
C

SOURCE CODE FOR CHANNEL MODEL

/* Advance survivor sequence memory by one step */

void advance_survivors()
{
    int i;
    int statesby2;
    BIT** survivors_new;

    statesby2 = num_states >> 1;

    /* Allocate memory for new set of survivors */
    survivors_new = (BIT**)calloc(num_states, sizeof(BIT*));

    for(i = 0; i < num_states; i++)
    {
        BIT* dest;
        BIT* src;

        /* Allocate memory for new survivor row */
        survivors_new[i] = (BIT*)calloc(survivor_len + statesby2, sizeof(BIT));

        survivors_new[i][0] = 1 << 2; /* Initial state values */

        /* Copy appropriate survivor state from old memory */
        dest = &survivors[i][0];

        /* Choose source state based on decision */
        if(Decisions[i] == 0)
            src = &survivors[i+1][0];
        else
            src = &survivors[i][0];

        /* Do the copy */
        memcpy(dest, src, (survivor_len-1) * sizeof(BIT));
    }

    /* Free up old memory */
    for(i = 0; i < num_states; i++)
        free(survivors[i]);

    free(survivors);

    /* Take possession of new memory */
    survivors = survivors_new;
}

#endif

int row, col;
print("Survivor memory (after update)\n");
for(row = 0; row < num_states; row++)
{
    for(col = 0; col < survivor_len; col++)
    {
        printf("%d", survivors[row][col]);
    }
    printf("\n");
}

if(getchar() == 'q') exit(10);
}

#endif

#endif

/* If defined(MERGE_DEPTH_TEST) */

void merge_depth(void)
{
    int i, j;
    int merged;

    i = 0;
    merged = -1;
    while(i < survivor_len && (merged == -1))
    {
        merged = i;

        for(j = 1; j < num_states; j++)
        {
            if(survivors[j][i] == survivors[j-1][i])
                merged = -1;
        }
        i++;
    }
}

return merged;
}

#endif

/* Define DEMUX */

int row, col;
print("Survivor memory (after update)\n");
for(row = 0; row < num_states; row++)
{
    if(is_valid_branch(row, 0) || is_valid_branch(row, 1))
    {
        printf("%d", row);
        for(col = 0; col < survivor_len; col++)
        {
            printf("%d", survivors[row][col]);
        }
        printf("\n");
    }
}

if(getchar() == 'q') exit(10);
}

#endif

}
This program takes in the branch metric values and channel data stream from the DVD read channel simulation, and uses a Viterbi decoder. It simulates to re-generate the original channel input in the presence of a user-controlled range of AWGN SNR. The results are compared to a reference file and the final BER is output.

Quantization is turned on/off by having the QUANTIZATION_ON macro defined at /COMPILE_TIME/.

/* AUTHOR */
Kenneth Chalmers
Department of Electrical and Computer Engineering
University of Toronto

/* COPYRIGHT */
Copyright 1999 Kenneth Chalmers

/*
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <malloc.h>
#include <math.h>
#include <string.h>
#include <limits.h>

#if defined(QUANTIZATION_ON)
#define QUANTIZATION is on
#include "viterbi_q.h"
#include "quantize.h"
#define Q_LOW -1.45 /* -1.0 for 2 state */
#define Q_RUSH -Q_LOW /* Symmetric about x axis */
#else
#define QUANTIZATION is off
#include "viterbi.h"
#endif

#include "noisy.h"
#include "prune_trail.h"
#include "edl_jitter.h"

#define SLICE(x) { x > 0 ? 1 : 0 }
#define LATENCY_TEST
#define NO_PUREParent
#define JITTER_ON

typedef enum { RESEARCH, TRACERBACK } SHU_METHOD;

/* Function prototypes */
int main(int ac, char** av);
int Viterbi(char* hmn, char* datfn, char* resultsfn, char* reffn, double SNRLOW, double SNRHI, double SNRSTEP, double jitter_pct, int qbits);
int GetBranchMetrics(char* hmn, BRANCH_METRIC* lm);
double SNR_cal(double* normal, double* noisy, int len);

/* Globals */
SHU_METHOD sum;
*/
*/
/* Main entry point */
int main(int ac, char** av)
{
    int qbits;
    if (ac < 10)
    {
        printf("Usage: viterbi <dat-file> <resultsfile> <reference file> <SNR LOW>
               <SNR HIGH> <SNR Step> <jitter %> <Quant. Bits in>");
    } 
    qbits = atof(av[9]);
    Q_BITS = qbits;
    if(ac < 9)
    {
        printf("Usage: viterbi <dat-file> <resultsfile> <reference file> <SNR LOW>
               <SNR HIGH> <SNR Step> <jitter %>");
    } 
    qbits = -1;
#endif
    return qbits;

    printf("Input file name same as an input file (branch metric or data) \n\n");
    printf("Overwrite input file? (YN) ");
    ch = getchar();
    if(ch<0 || ch>'Y') return 0;
    return 5;

    if(strcmp(ch, "y") == NULL)
    {
        printf("<address>, Traceback\n");
        sum = TRACERBACK;
    }
    else
    {
        printf("<address>, "Register exchange\n");
        sum = RESEARCH;
    }
    return Viterbi(av[1], av[2], av[3], av[4], atoi(av[5]), atoi(av[6]), atoi(av[7]), atoi(av[8]), qbits);
}
*/
*/
Viterbi tested. Runs Viterbi decoder on a data file saves results.*/
int Viterbi(char* hmn, char* datfn, char* resultsfn, char* reffn, double SNRLOW, double SNRHI, double SNRSTEP, double jitter_pct, int qbits)
FILE* data_f;  /* Input data for Viterbi */
FILE* results_f;  /* Output file for results */
FILE* ref_f;     /* Reference file (original data stream) for comparison */
long* branch_metric;  /* Branch metric array */
double* data_a;    /* Array of input data for Viterbi */
double* noisy_data_a;  /* Array with noisy version of data_a */
double true_err;    /* Calculated SNR after all impairments applied */
int* ref_a;        /* Array of reference (original data) values */
int numref;        /* Count of # of values in ref_a */
int numstates;     /* Number of entries in branch metric array */
int* state_numread;  /* Number of data items actually read (file size check) */
int INTRJT;       /* Input to Viterbi decoder */
int outdata;       /* Output from Viterbi decoder */
int vit_latency;   /* Latency of Viterbi decoder (for comparisons) */
int slice_latency; /* Latency for slicer */
int errcount = 0;  /* Error count of results vs. reference file */
int sliceerr = 0;  /* Error count for simple slicer */
int* refval;       /* Reference value read from ref file */
int datastates;    /* Size of data file */
double snr;        /* Current SNR in loop */
int count = 0;     /* Count of # of comparisons (output vs. ref) done */
int l;            /* Constraint length of Viterbi decoder */
int survivor_len;  /* Length of survivor sequence memory */

if((num = GetBranchMetrics(refm, slm)) == -1)
return 5; /* -1 return by GEN means error */

/* Calculate the number of states, 2*nu */
numstates = 1 << nu;

/* Calculate merge depth and latency based on constraint length */
switch(nu) {
  case 4:/* For 10 bits of input data ISI */
    /* Register exchange, survivor_len = 25 */
    vit_latency = 22;
    /* Note changed values for traceback! Won’t work with reacase any more */
    if(snum == TRACEBACK)
    {
      survivor_len = 27;
      /* latency = (2k(k-1)) * survivor_len + 13 */
      /* For k = 3 */
      vit_latency = ((survivor_len) + 13);
      /* For k = 4 */
      vit_latency = (2*survivor_len) + 13);
      /* Original data used to discover above formula */
      /* vit_latency = (3) */
      /* For k = 3, survivor_len = 26 */
      /* vit_latency = (3) */
      /* For k = 4, survivor_len = 26 */
      /* vit_latency = (3) */
      /* For k = 4, survivor_len = 26 */
    }
    slice_latency = vit_latency-11; /* For 10 bits of input data ISI */
    break;
    case 5:
    case 6:
      survivor_len = 30;
      vit_latency = 27;  /* For 10 bits of input data ISI */
      slice_latency = vit_latency-11; /* For 10 bits of input data ISI */
      break;
    case 7:
    case 8:
      survivor_len = 30;
      vit_latency = 27;  /* For 10 bits of input data ISI */
      slice_latency = vit_latency-11; /* For 10 bits of input data ISI */
      break;
    case 9:
      survivor_len = 40;
      vit_latency = 34;
      slice_latency = vit_latency-11; /* For 10 bits of input data ISI */
      break;
    case 10:
      survivor_len = 50;
      vit_latency = 44;
      slice_latency = vit_latency-11; /* For 10 bits of input data ISI */
      break;
    case 11:
    case 12:
      survivor_len = 50;
      vit_latency = 43;
      slice_latency = vit_latency-11; /* For 10 bits of input data ISI */
      break;
}
case 13:
case 14:
    survivor_len = 60;
vit_latency = 52;
slice_latency = vit_latency-11;
break;
#endif
default:
    fprintf(stderr, "Unknown # of states %d, no latency data, aborting\n", numstates);
    return 5;
#endif
#endif
#endif
}

if (defined(NO_PRUNE))
    /* Prune the trellis for run-length limit violations */
    prune_trellis(bm, nu, 2);
#endif
#endif

#endif

#if defined(QUANTIZATION_ON)
for(i = 0; i < numstates; i++)
    fprintf(stderr, " # %d:\%d\%d\%d\n", i, bm[i].metric_value[0], bm[i].metric_value[1]);
#endif
#endif

} 
#endif

/* Read in channel data - note binary file read */
data_t = (FILE*)fopen(datafn, "rb");
if(data_t == NULL) {
    printf("Error: could not open input data file %s\n", datafn);
    return 5;
}
#endif
#endif
#endif

/* Read number of elements */
if_read(datasize, sizeof(int), 1, data_t);
if(data_t == NULL) {
    printf("Not enough memory for input array\n");
    return -1;
}
#endif
#endif
#endif

/* Allocate array for input data */
data_a = (double*)malloc(sizeof(double) * datasize);
if(data_a == NULL) {
    printf("Not enough memory for input array\n");
    return -1;
}
#endif
#endif
#endif

/* Read array */
numread = fread(data_a, sizeof(data_a), datasize, data_t);
if(numread != datasize) {
    printf("Error: Data file truncated.\n");
    return 5;
}
#endif
#endif
#endif

fclose(data_t);
#endif
#endif
#endif

/* Start actual Viterbi test */
#endif
#endif
#endif

/* Allocate memory for noisy copy of input data */
noisy_data_a = (double*)malloc(sizeof(double) * datasize);
if(noisy_data_a == NULL) {
    printf("No memory for noisy data\n");
    return -1;
}
#endif
#endif
#endif

for(snr = SNRLow; snr < SNNHigh; snr += SNNStep) {
    /* Print the current SNR to stderr (simple action display) */
    fprintf(stderr, "SNR:
", snr);
    /* Make a copy of the input array */
    memcpy(noisy_data_a, data_a, sizeof(double) * datasize);
    if (defined(JITTER_ON))
        add_jitter(noisy_data_a, datasize, jitter_pct);
   #endif
    /* Noisify the copy according to the current SNR */
    noisify(noisy_data_a, datasize, snr);
    /* Calculate true SNR */
    true_snr = snr_calc(data_a, noisy_data_a, datasize);
    /* DEBUG: Print out first 5 data/noisy values */
    for(i = 0; i < 5; i++)
        printf("%.10f
", noisy_data_a[i]);
}
fclose(hm_f); return -1; }
}
for(i = 0; i < numstates; i++)
{
  if(feof(hm_f))
  { printf("Error: branch metric file is truncated\n");
    free(hm);
    close(hm_f);
    return -1;
  }
  fscanf(hm_f, "%g %g", &hm0, &hm1);
}
#else defined(QUANTIZATION_ON)
  /* Quantize the branch metrics */
  (*hm[i][].metric_value[0] = quantize(hm0);
   (*hm[i][].metric_value[1] = quantize(hm1);
#else /* Retrieve the branch metrics */
  (*hm[i][].metric_value[0] = hm0;
   (*hm[i][].metric_value[1] = hm1;
#endif
  
  /* Close the branch metric file */
  fclose(hm_f);
  return num; /* Successful exit */
}

double snr_calc(double* normal, double* noisy, int len)
{
  double noise, sig_total, noise_total;
  int i;
  sig_total = 0;
  noise_total = 0;
  for(i = 0; i < len; i++)
  {
    noise = noisy[i] - normal[i];
    sig_total += normal[i] * normal[i];
    noise_total += noise * noise;
  }
  return 10.0 * log10(sig_total/noise_total);
}

/*
 * snl.c
 *
 * A zero-memory nonlinearity (ZNL) function used to introduce nonlinear
 * distortion into the DVD channel simulation.
 */

/* REFERENCES */
Hwang, I.S. and Lee, Y.H., "Partial Response Equalization with
  Nonlinearity Compensating Signal Asymmetry in DVD Storage", in
  Optical Data Storage '98, Shigeku Kubota, Tom D. Miljuk, Paul J.

/* AUTHOR */
Kenneth Chalmers
Department of Electrical and Computer Engineering
University of Toronto

/* COPYRIGHT */
Copyright 1999 Kenneth Chalmers

#include <math.h>
#include "snl.h"

#ifndef ALPHA1
#define ALPHA1 0.597
#endif
#ifndef ALFA2
#define ALFA2 -0.456
#endif
#ifndef N1
#define N1 1.331
#endif
#ifndef N2
#define N2 0.642
#endif

#ifndef GAMMA1
#define GAMMA1 0.5*(GAMMA1*(1-N1) + GAMMA2*(1-N2))
#endif
#ifndef GAMMA2
#define GAMMA2 0.5*(N1 + N2)
#endif
#ifndef BETA1
#define BETA1 0.5*(1 - N1)
#endif
#ifndef BETA2
#define BETA2 0.5*(1 - N2)
#endif

double snl(double val)
{
  return ALPHA1 + (ALPHA2 * val) + BETA1*fabs(val - GAMMA1) + BETA2*fabs(val - GAMMA2);
}
Appendix C

VHDL Source Code
library ieee;
use work.globales.all;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_signed.all;

-- Entity declaration
entity aca is
port
 (   bm  : in BRANCH_METRIC_ARRAY;
    clk, reset : in std_logic;
    decisions : out DECISION_ARRAY
 );
end aca;

-- Architecture declaration
architecture aca_struc of aca is
component aca_block
port
 (   bm0, bm1, path_metric_in, path_metric_out : in INTERNAL_VECTOR;
    decision : out std_logic
 );
end component;
signal bm_in: BM INTERNAL;
signal pm_out, pm_in, pm_out_shift: PATH_METRIC_MEMORY;
signal decisions_shift: DECISION ARRAY;
begin
  -- Sequential bit; path metric memory
  SEQUENTIAL: process(clk, reset)
  begin
    if reset = '1' then
      for i in 0 to NUM_PM-1 loop
        pm_in[i] <= conv_std_logic_vector(0, INTERNAL_SIZE);
      end loop;
      decisions <= conv_std_logic_vector(0, NUM_DEC);
      elsif rising_edge(clk) then
        pm_in <= pm_out;
        decisions <= decisions_shift;
      end if;
    end process;

    -- Resize the branch metrics from INPUT_SIZE to INTERNAL_SIZE
    ORIGIN: process(mn, pm_in)
    begin
      for i in 0 to NUM_PM-1 loop
        pm_in[i](INPUT_SIZE downto 0) <= bm(i);
      end loop;
    end process;

    -- ACS trellis (note only 4 signals need ACS block, other 4 only have 1 input)
    ACS: component aca_block port map (bm_in, mn_in, pm_in, pm_in, pm_out_shift, decisions_shift);
  end component;
end process;

  -- Normalisation process
  NORMALIZATION: process(pm_out_shift)
  begin
    variable normalisation: std_logic;
    normalisation := not (pm_out_shift(0)(INTERNAL_SIZE-1) and
                           pm_out_shift(1)(INTERNAL_SIZE-1) and
                           pm_out_shift(2)(INTERNAL_SIZE-1) and
                           pm_out_shift(3)(INTERNAL_SIZE-1) and
                           pm_out_shift(4)(INTERNAL_SIZE-1) and
                           pm_out_shift(5)(INTERNAL_SIZE-1) and
                           pm_out_shift(6)(INTERNAL_SIZE-1) and
                           pm_out_shift(7)(INTERNAL_SIZE-1));

    -- Perform normalisation by adding NZB of each output with normalise
    for i in 0 to NUM_PM-1 loop
      pm_out[i](INTERNAL_SIZE-2 downto 0) <= pm_out_shift[i](INTERNAL_SIZE-2 downto 0);
    end loop;
  end process;
end aca_struc;

-- Configuration declaration
configuration aca_config of aca is
  for aca_struc
    -- for all; aca_block use configuration work.aca_block_config; and for;
end for;
end acs_config;

---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

-- acs_block
--
-- Single basic ACS node. Performs adding of branch metrics to path metrics and compare-select to choose smallest (branch metric + path metric) value.
--
-- AUTHOR
-- Kenneth Chalmers
-- Department of Electrical and Computer Engineering
-- University of Toronto
--
-- COPYRIGHT
-- Copyright 1999 Kenneth Chalmers.
--
-- REVISION HISTORY
-- Mar 12, 1999: Created file.
-- Mar 15, 1999: Updated description of ACS node.
-- Mar 16, 1999: Renamed from acs to acs_block.
-- Apr 06, 1999: Inlined comparator and multilog, removed components
library ieee;
use work.globals.all;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

-- Entity declaration
entity acs_block is
  port (hm0, hml, path_metric_in0, path_metric_in1, path_metric_out, decision : in internal_vector;
        path_metric_out : out internal_vector;
        decision : out std_logic);
end acs_block;

-- Architecture declaration
architecture acs_block_behav of acs_block is
begin
  path metric out <= sum0;
  decision <= '1';
end if;
end process;
end acs_block_behav;

-- Configuration declaration
configuration acs_block_config of acs_block is
  for acs_block_behav end for;
end acs_block_config;

---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

-- bmg
-- The Branch Metric Generator unit. Generates all outgoing branch metrics for all states. Note trellis used: M = 4 but only
-- 8 valid states, 12 valid edges, due to DAI restrictions.
--
-- AUTHOR
-- Kenneth Chalmers
-- Department of Electrical and Computer Engineering
-- University of Toronto
--
-- COPYRIGHT
-- Copyright 1999 Kenneth Chalmers.
--
-- REVISION HISTORY
-- Mar 16, 1999: Created file.
-- Mar 24, 1999: FIXED: Only one input, not array of inputs!
-- Mar 25, 1999: Added asynchronous reset
-- Apr 27, 1999: Changed to adapt to new bmg_block which has an output
-- of ITEM_VECTOR instead of ITEM_VECTOR.
-- May 24, 1999: Added code to register outputs, as per Synopsys
-- suggestions (blue binder p. 419). Note that this
-- increases the latency of the entire circuit by 1.
-- Removed above code - messes up somehow?
library ieee;
use work.globals.all;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

-- Entity declaration
entity bmg is
  port (input : in ITEM_VECTOR;
        hml, path_metric_in0, path_metric_in1, path_metric_out, decision : out ITEM_VECTOR);
end bmg;
-- Architecture declaration

architecture bm_block of bm is
component bm_block
port
begin
end component;

-- Signal assignments
signal bm_sig: BW_ARRAY;
signal input_sig: INPUT_VECTOR;
signal output_sig: BRANCH_METRIC_ARRAY;
begin
SEQ: process(clk, reset) begin
if reset = '1' then
  input_sig <= conv_std_logic_vector(0, INPUT_SIZE);
  output_sig <= conv_std_logic_vector(0, INPUT_SIZE);
elsif rising_edge(clk) then
  output_sig <= bm_sig;
  output_sig <= input;
-- Register outputs
end process;

-- Configuration declaration
configuration bm_config of bm is
for bm_block
  use work.hm.config;
end configuration;

end bm_block;

-- Basic building block (one input) of the branch metric generator

-- Author
-- Kenneth Chalmers
-- Department of Electrical and Computer Engineering
-- University of Toronto

-- Copyright
-- Copyright 1999 Kenneth Chalmers.

-- Revision history
-- Feb 03, 1999: Created file
-- Mar 18, 1999: Changed to add one extra bit to prevent overflow.
-- Circuit passed testbench test.
-- Apr 07, 1999: Renamed from bm_block to bm_block
-- Added hierarchy barrier and easier to type
-- Apr 27, 1999: Changed output from INTERNAL_VECTOR to INPUT_VECTOR
-- to fix Cadence synthesis problem
-- May 24, 1999: Added code to register outputs, as per Synopsys
-- suggestions (blue binder, p. 6-12). Note that this
-- increases the latency of the entire circuit by 1.

library ieee;
use work_globals.all;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_signed.all;

-- Entity declaration
entity bm_block is
  port
  (bmv,
    input : in INPUT_VECTOR;
    clk,
    reset : in std_logic;
    result : out INPUT_VECTOR)
end bm_block;

-- Architecture declaration
architecture bm_block is
signal bm_big, input_big, result_big: std_logic_vector(INPUT_SIZE downto 0);
begin
SEQ: process(clk, reset) begin
if reset = '1' then
  result_big <= conv_std_logic_vector(0, INPUT_SIZE);
elsif rising_edge(clk) then
  result_big <= result_big(INPUT_SIZE downto 0);
end if;
end process;

bm_big(INPUT_SIZE) <= '0';
bm_big(INPUT_SIZE downto 0) <= bmv;
input_big(INPUT_SIZE) <= '0';
input_big(INPUT_SIZE downto 0) <= input;
result_big <= abs(bm_big - input_big);
result_big <= result_big(INPUT_SIZE downto 0);
end bm_block;
-- Configuration declaration
configuration bmgb_config of bmgb block is
for bmgb behave and for;
end bmgb_config;

-- Entity declaration
entity bms is
port
| input : in INPUT_VECTOR;
| scan_bm, -- Flag: 1 = scan in branch metrics, 0 = normal operation
| clk;
| reset : in std_logic;
| bm_out : out BM_ARRAY
|);
end bms;

-- Architecture declaration
architecture bmgb_ctl of bmgb is
signal bm_vals : BM_ARRAY;
signal clk_scan : std_logic;
begin
-- If reset is high, scan in branch metrics through the input port
BEGIN
process(clk, reset) begin
if reset = '1' then
-- Get default branch metrics
bm_vals <= branch_metrics;
elif (rising_edge(clk)) then
-- Scan in a new branch metric value, ripple the rest down the chain
for i in NUM_BM-1 downto 0 loop
bm_vals(i) <= bm_vals(i-1);
end loop;
bm_vals(0) <= input;
end if;
end process;
clk_scan <= clk and scan_bm;
bm_out <= bm_vals;
-- Configuration declaration
configuration bmgb_config of bmgb is
for bmgb_ctl and for;
end bmgb_config;

-- Entity declaration
entity clk_dbl is
port
| sel, clk_dbl, clk_ph1, clk_ph2 : in std_logic;
| clk_out : out std_logic
|);
end clk_dbl;

-- Architecture declaration
architecture clk_dbl_ctl of clk_dbl is
begin
with sel select
library ieee;
use ieee.std_logic_1164.all;

package globs is
  -- Bit widths of inputs and internal lines
  constant INPUT_SIZE: integer := 5;
  constant INTERNAL_SIZE: integer := 8;

  -- Useful size counts
  constant NM_BK: integer := 12; -- Number of edges in trellis
  constant NM_DEC: integer := 4; -- Number of decision bits required
  constant NM_PM: integer := 8; -- Number of states in trellis

  -- Survivor memory
  constant SNU_DEPTH: integer := 25;

  -- Vectors using above widths
  subtype INPUT_VECTOR is std_logic_vector (INPUT_SIZE-1 downto 0);
  subtype SNU_VECTOR is std_logic_vector (INTERNAL_SIZE-1 downto 0);

  -- Useful types for input and internal lines
  type INPUT_VECTOR is array (0 to NM_PM-1) of std_logic;
  type INTERNAL_VECTOR is array (0 to NM_PM-1) of std_logic;
  type BRANCH_METRIC is array (0 to NM_PM-1) of std_logic;
  type INTERNAL_MTR is array (0 to NM_PM-1) of std_logic;
  type PATH_MTR is array (0 to NM_PM-1) of std_logic;

  -- Actual expected values for calculating branch metrics
  constant BRANCH_MTR : array (0 to NM_PM-1) of std_logic :=
    (others => "000001");

end globs;

-- AVHOR
-- Kenneth Chalmers
-- Department of Electrical and Computer Engineering
-- University of Toronto
-- COPYRIGHT
-- Copyright 1999 Kenneth Chalmers.
--
-- REVISION HISTORY
-- Feb 03, 1999: - Created file
-- Mar 16, 1999: - Added subtypes:
--   type INPUT_VECTOR, INTERNAL_VECTOR
-- Mar 17, 1999: - Added types:
--   INPUT_VECTOR, INTERNAL_VECTOR
-- Mar 24, 1999: - Added constants:
--   NM_BK, NM_DEC, NM_PM, SNU_DEPTH
--   subtype SNU_VECTOR, SNU_OUT_ARRAY
-- Apr 27, 1999: - Changed type of BRANCH_METRIC_ARRAY to
--   INPUT_VECTOR from INTERNAL_VECTOR
--   Added subtype INTERNAL_MTR
--   Added type INTERNAL_MTR equivalent to old
--   BRANCH_METRIC_ARRAY type

library ieee;
use work.globals.all;
use ieee.std_logic_1164.all;
use work.std_logic_arith.all;
use work.std_logic_unsigned.all;

-- Entity declaration

end;

entity smu is
port
(
  decisions : in DECISION_ARRAY;
  clk : in std_logic;
  out_bit : out std_logic
);
end smu;

-- Architecture declaration
architecture smu_struc of smu is
component smu_slice
port
(
  inputs : in smu_VECTOR;
  decisions : in DECISION_ARRAY;
  clk : in std_logic;
  outputs : out smu_VECTOR
);
end component;

signal out_sig: smu_OUT_ARRAY;
signal init: smu_VECTOR;
begin
init <= "01110001"; -- Special vector, highly dependent on pruned trellis
for i in 1 to smu_DEPTH-1 generate
  smu: smu_slice port map (init, decisions, clk, out_sig(0));
end generate;

out_sig <= out_sig(0);
end smu_struc;

-- Configuration declaration
configuration smu_conf of smu is
  for smu_struc
  -- for all: smu_slice use configuration work.smu_conf; and for;
  end for;
end smu_conf;

-- smu_slice
-- One slice of the survivor memory unit. Performs multiplexing input
-- to registers for those with multiple inputs.
--
-- AUTHOR
-- Kenneth Chalmers, University of Toronto
--
-- REVISION HISTORY
-- Mar 17, 1999: - Created file
-- Mar 23, 1999: - Updated entity vector sizes to use new constants
-- from globals.vhd
-- Mar 24, 1999: - Removed sel.sums component, now just use internal
-- process (should improve automatic synthesis).
-- Apr 06, 1999: - Removed mux_sums component, just inline all the with
-- statements (cuts down on # of components, should
-- make synthesis faster/easier)
-- Apr 07, 1999: - Renamed from smu_sums to smu_slice
-- (shows hierarchy better and easier to type)

library ieee;
use work.globals.all;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_signed.all;

-- Entity declaration
entity smu_slice is
port
(
  inputs : in smu_VECTOR;
  decisions : in DECISION_ARRAY;
  clk : in std_logic;
  outputs : out smu_VECTOR
);
end smu_slice;

-- Architecture declaration
architecture smu_struc of smu_slice is
signal mux0_out, mux1_out, mux2_out, mux3_out: std_logic;
begin
-- Multiplexers
with decisions(0) select
  mux0_out <= inputs(0) when '0',
              inputs(4) when others;
with decisions(1) select
  mux1_out <= inputs(0) when '0',
              inputs(4) when others;
with decisions(2) select
  mux2_out <= inputs(3) when '0',
              inputs(7) when others;
with decisions(3) select
  mux3_out <= inputs(3) when '0',
              inputs(7) when others;
end select;

SEQ: process(clk)
begin
  if rising_edge(clk) then
    outputs(0) <= mux0_out;
    outputs(1) <= mux1_out;
    outputs(2) <= inputs(1);
    outputs(3) <= inputs(2);
    outputs(4) <= inputs(3);
    outputs(5) <= inputs(4);
    outputs(6) <= mux2_out;
    outputs(7) <= mux3_out;
  end if;
end process;
end smu_struc;

-- Configuration declaration
configuration smu_config of smu_slice is
for smu_struc end for;
end smu_config;

-- viterbi
--
-- The Viterbi detector. Actually just a design connecting all the
-- various pieces, defined elsewhere, together.
--
-- AUTHOR
-- Kenneth Chalmers
-- Department of Electrical and Computer Engineering
-- University of Toronto
--
-- COPYRIGHT
-- Copyright 1999 Kenneth Chalmers.
--
-- REVISION HISTORY
-- Mar 24, 1999: - Created file
-- Apr 05, 1999: - Due to re-shuffling, use ACS instead of ACSU
-- May 04, 1999: - Added scan signal inputs/outputs
-- May 21, 1999: - Added clock doubler
-- May 24, 1999: - Moved clock doubler to new files (see clk_dbl.vhd and
-- viterbi_cd.vhd). Cleans up this design a bit and
-- simplifies Synopsys synthesis (synth this,
-- set_dont_touch, synth viterbi_cd).
-- May 27, 1999: - Added registers and scan-in for branch metric - allows
-- dynamic changing of branch metrics.
-- - Removed output test_so - integrates scan out with output.
-- - Created component lmg, and moved metric scan in there.
-- - Jun 01, 1999: - Removed test_ai and test_so - they were messing
-- Synopsys up when the scan chain was added as part of
-- a larger design (i.e. viterbi_eval.vhd). Check out
-- viterbi_standalone.vhd for one that can be synthesised
-- by itself
--
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
-- Entity declaration
entity viterbi is
port
( input : in INPUT VECTOR;
  scan_bm, -- Flag: 1 = scan in branch metrics, 0 = normal operation
  clk,
  reset : in std_logic;
  output : out std_logic
);
end viterbi;

architecture viterbi_struc of viterbi is
component lmg
port
( input : in INPUT VECTOR;
  bm, -- in BRANCH_METRIC ARRAY;
  clk,
  reset : in std_logic;
  output : out BRANCH_METRIC_ARRAY
);
end component;

component smu
port
( input : in INPUT VECTOR;
  bm, -- in BRANCH_METRIC ARRAY;
  clk,
  reset : in std_logic;
  output : out std_logic
);
end component;

signal bm_vals: BM ARRAY;
signal lmg_out: BRANCH_METRIC ARRAY;
signal smu_out: DECISION ARRAY;
begin
-- Branch metric storage manager (allows scan in etc.)
lmg: lmg port map(input, scan_bm, clk, reset, bm_vals);
-- Branch metric generator
lmg0: lmg port map(input, bm_vals, clk, reset, lmg_out);
-- Add-compare-select unit
acs: acs port map(acs_out, clk, reset, acs_out);
-- Survivor memory unit
smu0: smu port map(acs_out, clk, output);
end viterbi_struc;
-- Configuration declaration
configuration viterbi_config of viterbi is
   for viterbi_struc
   -- for all: bg use configuration work.bmg_config; and for:
   -- for all: wa use configuration work.wa_config; and for:
   -- for all: smu use configuration work.smu_config; and for:
   end for;
end viterbi_config;

------------------------------------------------------------------
------------------------------------------------------------------
-- viterbi_cd
--
-- The Viterbi detector with clock doubling.
--
-- AUTHOR
-- Kenneth Chalmers
-- Department of Electrical and Computer Engineering
-- University of Toronto
--
-- COPYRIGHT
-- Copyright 1999 Kenneth Chalmers.
--
-- REVISION HISTORY
-- May 24, 1999: - Created file
-- May 27, 1999: - Modified to reflect changes in viterbi.vhdl
-- - Moved swinging I/O to viterbi.vhdl

library ieee;
use work.globals.all;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

-- Entity declaration
entity viterbi_cd is
   port ( input,
          input_b : in INPUT_VECTOR; -- For swinging buffer
          scan_m,
          test_se, -- Scan enable
          test_s1, -- Scan in
          clk,
          clk_phi1, clk_phi2, -- Out-of-phase clocks for doubling
          reset : in std_logic;
          output,
          output_b : out std_logic -- For swinging buffer
          -- Total: 18 pins
          );
end viterbi_cd;

-- Architecture declaration
architecture viterbi_cd_struc of viterbi_cd is
component viterbi_sw
   port ( input,
          input_b : in INPUT_VECTOR; -- For swinging buffer
          scan_m,
          test_se, -- Scan enable
          test_s1, -- Scan in
          clk,
          reset : in std_logic;
          output,
          output_b : out std_logic -- For swinging buffer
          -- Total: 17 pins
          );
end component;
signal clk: std_logic;
signal input_sig: INPUT_VECTOR;
begin
   -- Clock doubler
   clk <= clk_phi1 xor clk_phi2;

   -- Viterbi detector
   vsw0: viterbi_sw port map(input, input_b, scan_m, test_se, test_s1,
                            clk, reset, output, output_b);
end viterbi_cd_struc;

-- Configuration declaration
configuration viterbi_cd_config of viterbi_cd is
   for viterbi_cd_struc and for:
end viterbi_cd_config;

------------------------------------------------------------------
------------------------------------------------------------------
-- viterbi_sw
--
-- The Viterbi detector with 2-input/output swinging buffers
--
-- AUTHOR
-- Kenneth Chalmers
-- Department of Electrical and Computer Engineering
-- University of Toronto
--
-- COPYRIGHT
-- Copyright 1999 Kenneth Chalmers.
--
-- REVISION HISTORY
-- May 24, 1999: - Created file
-- May 27, 1999: - Modified to reflect changes in viterbi.vhdl
-- May 31, 1999: - Added swinging input/output buffer
-- - Renamed to viterbi_sw.vhdl, moved clock doubler outside
-- (may now I can finally create the darn test vectors)

library ieee;
use work.globals.all;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

-- Entity declaration
entity viterbi_sw is
port
(input,
input_b : in INPUT_VECTOR; -- For swinging buffer
scan_bm,
test_se, -- Scan enable
test_si, -- Scan in
clk,
reset : in std_logic;
output,
output_b : out std_logic; -- For swinging buffer
output2: out std_logic;
end)
; -- Total: 17 pins
end viterbi_sw;

-- Architecture declaration
architecture viterbi_sw_arch of viterbi_sw is
component viterbi
port
(input : in INPUT_VECTOR;
scan_bm, -- Flag: 1 = scan in branch metrics, 0 = normal operation
clk,
reset : in std_logic;
output, output2: out std_logic
);
end component;

signal input_g: INPUT_VECTOR;
signal which_buffer: std_logic;
signal vit_out, output_g, output_b_g: std_logic;
begin
-- Viterbi detector
vit0: viterbi port map (input_g, scan_bm, clk, reset, vit_out);

-- Swinging buffer: handle 2 inputs over 2 cycles
SBUP: process(clk, reset) begin
if reset = '1' then
input_g <= conv_std_logic_vector('0', INPUT_SIZE);
which_buffer <= '0';
else if rising_edge(clk) then
if which_buffer = '0' then
input_g <= input;
output_g <= vit_out;
which_buffer <= '1';
else
input_g <= input_b;
output_b_g <= vit_out;
which_buffer <= '0';
end if;
end if;
end process;

output <= output_g;
output_b <= output_b_g;
end viterbi_sw_arch;

-- Configuration declaration
configuration viterbi_sw_config of viterbi_sw is
for viterbi_sw_arch and for;
end viterbi_sw_config;